

# P.O.O. avancée

## Sommaire

1	Les entrées/sorties .....	6
1.1	Manipulation des fichiers .....	6
1.2	Manipulation des flux.....	8
1.2.1	Flux prédéfinis .....	8
1.2.2	Classes de flux de données.....	10
1.2.3	Flux binaires (flux d'octets) .....	11
1.2.3.1	Flux binaires d'entrée .....	12
1.2.3.1.1	La classe abstraite <b>InputStream</b> .....	12
1.2.3.1.2	Flux binaires de communication en entrée.....	13
1.2.3.1.3	Flux binaire de traitement en entrée .....	14
1.2.3.1.4	Exemple de lecture dans un fichier .....	14
1.2.3.2	Flux binaires de sortie .....	15
1.2.3.2.1	La classe abstraite <b>OutputStream</b> .....	15
1.2.3.2.2	Flux binaires de communication en sortie .....	15
1.2.3.2.3	Flux binaires de traitement en sortie .....	15
1.2.4	Flux de caractères.....	16
1.2.4.1	Flux de caractères en entrée .....	17
1.2.4.1.1	La classe abstraite <b>Reader</b> .....	17
1.2.4.1.2	Flux de caractères pour la communication en entrée .....	17
1.2.4.2	Flux de caractères en sortie.....	18
1.2.4.2.1	La classe abstraite <b>Writer</b> .....	18
1.2.4.2.2	Flux de caractères pour la communication en sortie .....	18
1.2.5	Utilisation des flux .....	19
1.2.6	Exemple complet de manipulation des flux : copie d'un fichier .....	19
1.3	Compression.....	21
1.3.1	Compression .....	21
1.3.2	Décompression.....	23
1.4	Sérialisation .....	24
1.4.1	Vue globale de la persistance en Java .....	24
1.4.2	Mécanisme de sérialisation en Java .....	25
1.4.2.1	Interface et flux de sérialisation .....	25

1.4.2.2	Problèmes dus à la sérialisation .....	26
1.4.2.3	Sérialisation et héritage.....	27
1.4.2.4	Mise en œuvre.....	27
1.4.2.5	Exemples de sérialisation .....	28
1.4.2.5.1	Exemple de classe persistante .....	28
1.4.2.5.2	Exemple de classe sérialisant une classe persistante .....	29
1.4.2.5.3	Exemple de classe désérialisant une classe persistante .....	29
1.4.2.5.4	Exemple complet de sérialisation/désérialisation .....	30
1.4.2.6	Une précaution à prendre .....	32
2	Manipulation des flux sur sockets : communications réseau .....	33
2.1	Concept de socket .....	33
2.2	Les paquetages de Java dédiés.....	34
2.3	La classe <code>java.net.InetAddress</code> .....	34
2.4	Utilisation des sockets en mode connecté (TCP) .....	34
2.4.1	Du côté du serveur ( <code>java.net.ServerSocket</code> ).....	35
2.4.2	Du côté du client ( <code>java.net.Socket</code> ) .....	36
2.4.3	Flux utilisés .....	38
2.4.4	Exemple complet de client/serveur en mode connecté (TCP) .....	39
2.4.4.1	Du côté du serveur .....	39
2.4.4.2	Du côté du client .....	40
2.4.5	Les exceptions .....	40
2.5	Sockets en mode datagramme (UDP) .....	41
3	Gestion programmatique des classes .....	45
3.1	Réification.....	45
3.2	Intérêt et utilisation.....	45
3.3	Outils de mise en œuvre .....	46
3.4	Chargement dynamique des classes .....	46
3.4.1	Première solution de chargement dynamique de classes .....	47
3.4.2	Seconde solution de chargement dynamique de classes.....	47
3.5	Exemples d'utilisation .....	48
3.5.1	Exemple n°1.....	48
3.5.2	Exemple n°2.....	48
4	Processus concurrents.....	51

4.1	Les <i>threads</i> en Java .....	51
4.2	La classe <code>Thread</code> .....	52
4.2.1	Activer un <i>thread</i> .....	53
4.2.2	Arrêter un <i>thread</i> .....	53
4.2.3	Suspendre un <i>thread</i> .....	54
4.2.4	Attendre la fin d'un <i>thread</i> .....	55
4.2.5	Tester l'exécution d'un <i>thread</i> .....	56
4.2.6	Priorité d'exécution d'un <i>thread</i> .....	56
4.3	L'interface <code>Runnable</code> .....	58
4.3.1	Créer un <i>thread</i> via un objet <code>Runnable</code> .....	58
4.3.2	Accès aux <i>threads</i> créés à partir d'objets <code>Runnable</code> .....	61
4.4	Choisir la classe <code>Thread</code> ou l'interface <code>Runnable</code> ? .....	62
4.5	Cycle de vie d'un <i>thread</i> .....	62
4.6	Groupe de <i>threads</i> .....	63
4.7	Partage d'informations entre <i>threads</i> .....	64
4.7.1	Partage d'informations via des variables statiques .....	64
4.7.2	Partage d'informations via un objet <code>Runnable</code> .....	65
4.7.3	Comparaison des techniques de partage d'informations .....	66
4.8	Synchronisation/verrouillage .....	67
4.8.1	Synchronisation d'un bloc d'instructions .....	67
4.8.2	Synchronisation de méthodes .....	71
4.9	Mécanisme de notification entre <i>threads</i> .....	72
4.9.1	Attendre une notification : la méthode <code>wait()</code> .....	72
4.9.2	Notifier un <i>thread</i> : la méthode <code>notify()</code> .....	73
4.9.3	Notifier tous les <i>threads</i> : la méthode <code>notifyAll()</code> .....	73
4.9.4	Exemples de notification .....	73
4.9.5	Interblocage .....	76
4.9.6	Exemple de producteurs/consommateurs .....	78
5	Les modèles d'architecture .....	83
5.1	Exemples de 2 patrons de création .....	83
5.1.1	Abstract Factory .....	83
5.1.2	Singleton .....	87
5.2	Exemples de 2 patrons de structure .....	89

5.2.1	Adapter .....	89
5.2.2	Composite.....	91
5.3	Exemples de 2 patrons de comportement.....	95
5.3.1	Iterator .....	95
5.3.2	State.....	102

## 1 Les entrées/sorties

Comme tout bon langage de programmation, Java fournit une API de manipulation de flux de données. Les classes de cette API se répartissent dans les paquetages suivants :

- `java.io.*` pour les autres opérations d'entrées/sorties,
- `java.net.*` pour la manipulation des flux sur sockets.

### 1.1 Manipulation des fichiers

Les opérations de manipulation de fichiers se font au moyen de la classe `java.io.File`. Cette classe permet de renommer un fichier, de le supprimer, d'en connaître les droits d'accès, ...



#### Remarque

Il s'agit d'étudier ici la manipulation des fichiers et non pas des données qu'ils contiennent (ceci sera fait *via* la manipulation des flux).



#### Exemple

```
File f = new File("fichier.mp3");

System.out.println(f.getAbsolutePath() + f.getName());

if (f.exists())
{
    System.out.println(f.getName() + " : " +
        (f.canRead() ? "r" : "-") +
        (f.canWrite() ? "w" : "-") + " : " +
        f.length());
    f.delete();
}
```

La classe **File** est une représentation abstraite de tout « fichier » (fichiers, répertoires, liens symboliques, ...). Cette représentation est dite « abstraite » car le fichier représenté n'existe pas forcément. Par la suite, on utilisera le terme « fichier » pour désigner un fichier aussi bien qu'un répertoire ou qu'un lien symbolique, ...

Cette classe est très utile pour obtenir des informations sur un fichier ou un répertoire.

En utilisant la propriété système **File.separator** (de manière transparente pour le développeur), vous pouvez créer des fichiers dans un répertoire sans avoir à vous soucier du type de système sur lequel l'application sera exécutée.

La classe **File** possède 4 constructeurs :

- **File(String pathname)** : ce constructeur prend en paramètre une chaîne de caractères (**String**) correspondant au chemin (relatif ou absolu) du fichier.
- **File(String parent, String child)** : ce constructeur prend en paramètres deux chaînes de caractères, à savoir le chemin du répertoire parent et le nom du fichier.
- **File(File parent, String child)** : comme pour le constructeur, celui-ci prend deux paramètres, à savoir le fichier représentant le répertoire parent et le nom du fichier.
- **File(URI uri)** : ici le constructeur prend en paramètre l'URI du fichier convertie en un chemin abstrait.



#### Attention

En instanciant la classe **File**, vous ne créez pas physiquement un fichier mais une représentation d'un fichier (que celui-ci existe ou non).

Principales méthodes de cette classe :

- **boolean canRead()** : permet de savoir s'il est possible lire le fichier (possession de droits en lecture).
- **boolean canWrite()** : permet de savoir s'il est possible d'écrire dans le fichier (possession de droits en écriture).
- **boolean createNewFile()** : permet de créer (création physique) le fichier. Renvoie **false** si le fichier existe déjà.
- **boolean delete()** : supprime (physiquement) le fichier.
- **boolean exists()** : teste si le fichier existe (physiquement).
- **String getAbsolutePath()** : retourne une chaîne de caractères correspondant au chemin absolu du fichier.
- **String getName()** : retourne une chaîne de caractères correspondant au nom du fichier.
- **File getParentFile()** : retourne un **File** correspondant au répertoire parent.
- **boolean isDirectory()** : teste si le fichier est un répertoire
- **boolean isFile()** : teste si le fichier est un fichier.
- **String[] list()** : retourne un tableau de **String** contenant les noms des fichiers du répertoire. Cette méthode peut prendre en paramètre un **FilenameFilter** pour filtrer les fichiers retournés.
- **File[] listFiles()** : retourne un tableau de **File** contenant les **File** représentant les fichiers du répertoire. Cette méthode peut prendre en paramètre un **FilenameFilter** pour filtrer les fichiers retournés.
- **boolean mkdir()** : crée le répertoire représenté par le **File**.
- **boolean mkdirs()** : crée le répertoire représenté par le **File** avec les répertoires parents s'ils n'existent pas.

**Exemple**

Affichage du contenu d'un répertoire



```
import java.io.*;

public class Scan
{
    public static void main(String[] args)
        throws Exception
    {
        if (args.length != 1)
        {
            System.out.println
                ("Saisir un dossier !");
            System.exit(0);
        }

        File f = new File(args[0]);
        String [] files = f.list();

        for (int i =0; i < files.length; i++)
        {
            if (new File(args[0] + "\\\" +
                files[i]).isDirectory())
            { System.out.print("Rep : "); }
            else
            { System.out.print("Fic : "); }

            System.out.println(args[0] + "\\\" +
                files[i]);
        }
    }
}
```

## 1.2 Manipulation des flux

Il existe plusieurs types de flux :

- Basés sur des fichiers,
- Basés sur des sockets réseau,
- Basés sur la console d'applications,
- ...

Leur utilisation en Java est similaire.

### 1.2.1 Flux prédéfinis

Il existe 3 flux prédéfinis :

- **System.in** (instance de la classe **InputStream**) : l'entrée standard,
- **System.out** (instance de la classe **PrintStream**) : la sortie standard,
- **System.err** (instance de la classe **PrintStream**) : la sortie standard d'erreurs.



**Exemple**

Utilisation des flux prédéfinis



```
try
{
    int c, nbc = 0;

    while ((c = System.in.read()) != -1)
    {
        System.out.print(c);
        nbc++;
    }
}
catch (IOException exc)
{
    exc.printStackTrace();
    // En fait exc.printStackTrace(System.err);
}
```

Pour ce qui est de la manipulation du flux de sortie, il est préférable de ne pas directement utiliser la classe **InputStream**. En effet, elle ne propose que des méthodes élémentaires de récupération de données. Préférez la classe **BufferedReader**.

Nous reviendrons ultérieurement sur la notion de **Reader** et de **Writer**. Pour l'heure, sachez simplement générer un objet **BufferedReader** à partir de **System.in**, comme le montre l'exemple suivant. Cela vous permettra de facilement récupérer des chaînes de caractères saisies sur la console par l'utilisateur.

**Exemple**Utilisation (préférable) de **BufferedReader** pour traiter les saisies sur la console.

```
Reader reader = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(reader);

System.out.print("Entrez une ligne de texte : ");
String line = keyboard.readLine();
System.out.println("Vous avez saisi : " + line);
```

## 1.2.2 Classes de flux de données

Ces classes sont réparties comme suit, en fonction de leur « direction » (entrée ou sortie) et du type de données qu'elles manipulent (octets ou caractères).

	Flux d'entrée	Flux de sortie
<b>JDK 1.0</b> <b>Flux d'octets</b> <b>(8 bits)</b>	InputStream <ul style="list-style-type: none"> <li>• FileInputStream</li> <li>• DataInputStream</li> <li>• BufferedInputStream</li> <li>• ...</li> </ul>	OutputStream <ul style="list-style-type: none"> <li>• FileOutputStream</li> <li>• DataOutputStream</li> <li>• BufferedOutputStream</li> <li>• ...</li> </ul>
<b>JDK 1.1+</b> <b>Flux de caractères</b> <b>(16 bits)</b>	Reader <ul style="list-style-type: none"> <li>• FileReader</li> <li>• BufferedReader</li> <li>• StringReader</li> <li>• ...</li> </ul>	Writer <ul style="list-style-type: none"> <li>• FileWriter</li> <li>• BufferedWriter</li> <li>• StringWriter</li> <li>• ...</li> </ul>

Tableau 1 : classification des principales classes de flux en Java

### Question

Quand utiliser les flux binaires (flux d'octets) et quand utiliser les flux de caractères ?



Les « streams » (flux d'octets) utilisent toujours un codage en 8 bits, ce qui a rapidement posé des problèmes, tout simplement parce que la plupart du temps le contenu d'un fichier est codé en 16 bits. Les « readers/writers » (flux de caractères), eux, permettent de réaliser la transformation d'un code Unicode (16 bits) vers un système ASCII dérivé (8 bits) et vice-versa. Donc, quand on fera de la manipulation de flux, sans nécessité de prendre en compte le codage des caractères stockés dans le flux, on utilisera les flux binaires, plus performants. Par contre, quand on fera de la manipulation de flux avec une nécessité de prendre en compte le codage des caractères stockés dans le flux, on utilisera des flux de caractères. Ainsi, typiquement, réaliser une copie intégrale du contenu d'un fichier se fera *via* des flux binaires alors que l'affichage à l'écran du contenu d'un fichier se fera *via* des flux de caractères.

Deux classes permettent de faire la transition des « streams » vers des « readers/writers » :

- **InputStreamReader** permet de transformer un **InputStream** en un **Reader**,
- **OutputStreamWriter** permet de transformer un **OutputStream** en un **Writer**.

### Exemple

Encapsulation d'un flux d'octets (**System.in**) dans un flux de caractères (**keyboard**) en utilisant **InputStreamReader**.



```
Reader reader = new InputStreamReader(System.in);
BufferedReader keyboard = new BufferedReader(reader);

System.out.print("Entrez une ligne de texte : ");
String line = keyboard.readLine();
System.out.println("Vous avez saisi : " + line);
```

### 1.2.3 Flux binaires (flux d'octets)

Les flux binaires, répartis en flux binaires d'entrée et flux binaires de sortie, sont pris en charge *via* un ensemble de classes :

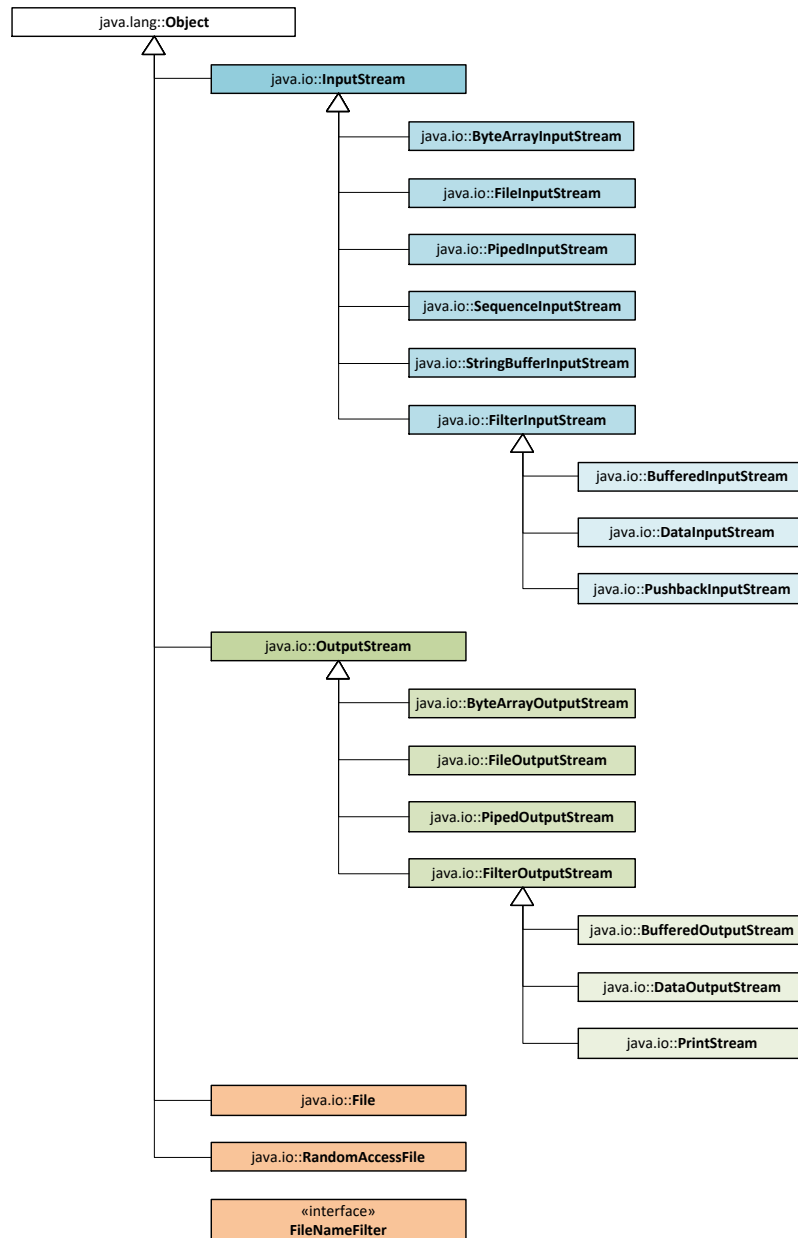


Figure 1 : classes prenant en charge les manipulations de flux binaires

On remarque que cette arborescence se divise en deux grandes familles :

- Les flux d'entrée (héritant d'**InputStream**),
- Et les flux de sortie (héritant d'**OutputStream**).

Les classes ayant le suffixe **InputStream** servent pour la lecture de données, tandis que celles ayant pour suffixe **OutputStream** permettent l'écriture. Les sous-classes directes de **InputStream** et **OutputStream** sont des flux de communications : elles permettent de lire et d'écrire des données.

Les classes **FilterInputStream** et **FilterOutputStream** sont les super classes des flux de traitements. Elles font un traitement sur les données lues ou écrites *via* le flux de communication auquel elles sont connectées.

L'interface **FilenameFilter** sert de filtre sur les noms de fichier. Elle possède la méthode (à redéfinir dans ses implémentations) **public boolean accept(File dir, String name)** qui sert à filtrer le contenu d'un répertoire. Celle-ci renvoie **true** si le fichier appelé **name** peut-être inclus dans une liste de fichiers.

La classe **RandomAccessFile** supporte la lecture et l'écriture de manière directe (et non séquentiellement) vers un fichier à accès aléatoire. Ce dernier se comporte comme un grand tableau d'octets stocké dans le système de fichiers.

La classe **File** est une représentation « abstraite » d'un fichier au sens UNIX du terme : fichier ou répertoire. Cette classe **File** n'est cependant elle-même pas abstraite (cf. plus haut).

### 1.2.3.1 Flux binaires d'entrée

Les objets permettant la lecture de flux de données binaires héritent tous de la classe abstraite **InputStream**. Les différents objets héritant d'**InputStream** fournissent des méthodes permettant la lecture séquentielle (on se déplace dans le flux) d'octets.

#### 1.2.3.1.1 La classe abstraite **InputStream**

**InputStream** est la super classe de toutes celles représentant un flux d'entrée d'octets. Chaque sous-classe doit redéfinir une méthode **read()** retournant le prochain octet sur le flux d'entrée. À chaque appel de cette méthode, on se déplace d'un octet sur le flux. Bien qu'un octet (8 bits) corresponde à un **unsigned byte** en Java, la méthode **read()** retourne un **int** (32 bits). En effet, on a plus l'habitude de travailler avec des entiers qu'avec des octets. De plus, la conversion ne pose aucun problème, un **int** étant « plus grand » qu'un **byte**. Si on a atteint la fin du flux et que l'on ne peut plus lire, la méthode retourne la valeur -1. La classe fournit également deux autres méthodes **read()** :

- **int read(byte[] b)** : avec cette méthode, vous pouvez lire plus d'un octet à la fois. En effet, les octets lus sont stockés dans le tableau passé en paramètre, remplaçant les éventuelles anciennes valeurs. La méthode lit autant d'octets qu'elle peut en stocker dans le tableau et autant qu'elle peut en lire sur le flux. La méthode retourne le nombre réel d'octets lus et -1 si la lecture a échoué (fin du flux).
- **int read(byte[] b, int off, int len)** : le principe est similaire à la méthode précédente, cependant au lieu de remplir le tableau dès la première position, on commence à écrire les octets à la position **off** (offset) du tableau. Le paramètre **len** correspond au nombre maximum d'octets à lire. Faites donc attention à ce que votre tableau ait bien la taille nécessaire. Comme pour la méthode précédente, l'entier retourné correspond au nombre réel d'octets lus et -1 si la lecture a échoué (fin du flux).

Cette classe fournit également d'autres méthodes utiles à l'ensemble de ses sous-classes :

- **int available()** : cette méthode retourne le nombre d'octets qui peuvent être lus (ou passés) sur le flux d'entrée sans bloquer la prochaine lecture.
- **close()** : ferme le flux et libère toutes les ressources systèmes liées à ce flux. Cette méthode doit toujours être appelée une fois que vous avez fini de travailler avec le flux.

Par défaut, la lecture dans un flux se fait de façon séquentielle. Néanmoins, certaines méthodes permettent ponctuellement d'outrepasser ce mode de lecture :

- **mark(int readLimit)** : marque la position courante dans le flux. Le prochain appel de la méthode **reset()** repositionne le flux à cette position, ainsi la prochaine lecture relira les octets à partir de la marque posée. Le paramètre **readLimit** spécifie le nombre d'octets pouvant être lus. Lorsque cette limite est atteinte la marque disparaît.
- **reset()** : repositionne le flux à la dernière position définie par la méthode **mark()**.
- **boolean markSupported()** : teste si le flux d'entrée supporte les méthodes **mark()** et **reset()**.
- **skip(long n)** : permet de passer et d'ignorer **n** octets de données sur le flux.

#### 1.2.3.1.2 Flux binaires de communication en entrée

Les sous-classes d'**InputStream** sont des flux de communication. Elles permettent une lecture séquentielle d'octets du flux spécifique qu'elles représentent. Voici les principales classes héritant d'**InputStream** :

- **ByteArrayInputStream** : possède un buffer interne (passé en paramètre du constructeur) qui peut être lu à partir du flux. Un compteur interne garde une trace du prochain octet à lire. À noter que la méthode **close()** n'a aucun effet.
- **FileInputStream** : permet la lecture des données contenues dans un fichier. Cette classe est utilisée, par exemple, pour lire une image.
- **ObjectInputStream** : cette classe sert pour la désérialisation. Elle possède différentes méthodes pour lire tout type de données (primitifs ou objets).
- **PipedInputStream** : cette classe est largement inspirée du *pipe* sous les systèmes UNIX (la sortie d'une instruction correspond à l'entrée d'une autre instruction). Elle travaille de paire avec un **PipedOutputStream** auquel elle est connectée. La connexion peut être effectuée directement à la création de l'objet (*via* le constructeur) ou par la suite grâce à la méthode **connect()**. Typiquement, les données sont lues à partir d'un **PipedInputStream** par un *thread* et écrites sur le **PipedOutputStream** par un autre *thread*.
- **SequenceInputStream** : représente la concaténation logique d'autres flux de lecture. Lorsqu'on atteint la fin d'un flux, la lecture continue sur le flux suivant. Les flux de lecture sont passés au constructeur sous forme d'**Enumeration**.
- **StringBufferInputStream** : cette classe et l'ensemble de ses méthodes sont dépréciées car convertissent mal les caractères en octets (**bytes**), nous ne l'étudierons donc pas. Si vous avez besoin de créer un flux à partir d'une chaîne de caractères, vous pouvez utiliser la classe **StringReader**, qui hérite de **Reader** (voir plus loin).

### 1.2.3.1.3 Flux binaire de traitement en entrée

Les classes présentées ici sont des sous-classes de **FilterInputStream**. Elles font un traitement sur les données lues sur un flux de communication associé. Le flux de communication (**InputStream**) est passé en paramètre au constructeur.

- **BufferedInputStream**: cette classe possède un buffer interne. Elle ajoute, au flux associé, une mise en mémoire tampon lors de la lecture. Elle permet également au flux de communication de supporter les méthodes **mark()** et **reset()**.
- **DataInputStream**: permet la lecture de données sous forme de type primitif. Une application écrit des données *via* un **DataOutputStream** qui pourront être lues par la suite grâce à un **DataInputStream**. Ces deux flux représentent des chaînes de caractères au format Unicode.
- **PushbackInputStream**: cette classe ajoute la possibilité de revenir en arrière dans un flux. Il sera alors possible de lire certains octets plusieurs fois. Cela peut être utile si vous cherchez un ou des caractères de séparation qui peuvent être ignorés dans certains cas.

### 1.2.3.1.4 Exemple de lecture dans un fichier

#### Exemple



```
import java.io.*;

public class BinaryReaderInFile
{
    public static void main(String[] args)
    {
        byte[] buffer = new byte[53];

        try
        {
            FileInputStream fis =
                new FileInputStream("C:\\java.gif");
            int i = fis.read(buffer);
            String s = new String(buffer);
            System.out.println(s);
            System.out.println(i);
        }
        catch (FileNotFoundException e)
        { System.err.println("Fichier non trouvé."); }
        catch (IOException e)
        { System.err.println("Impossible de lire."); }
    }
}
```

### 1.2.3.2 Flux binaires de sortie

Les objets permettant l'écriture dans un flux de données binaires héritent tous de la classe abstraite **OutputStream**.

#### 1.2.3.2.1 La classe abstraite **OutputStream**

**OutputStream** est la super classe de toutes celles représentant un flux de sortie d'octets. Chaque sous-classe doit toujours redéfinir au moins une méthode **write(int b)** prenant en paramètre un octet (bien que contenu dans un **int**) à écrire sur le flux de sortie. La classe fournit également deux autres méthodes **write()** :

- **void write(byte[] b)** : avec cette méthode, vous pouvez écrire plus d'un octet à la fois. En effet, les octets stockés dans le tableau passé en paramètre seront tous écrits en un seul appel de cette méthode.
- **void write(byte[] b, int off, int len)** : le principe est similaire à la méthode précédente, cependant au lieu d'écrire sur le flux l'intégralité du tableau, on commence à écrire les octets à partir de la position **off** (offset) du tableau. Le paramètre **len** correspond au nombre d'octets à écrire.

De plus, elle fournit deux autres méthodes :

- **close()** : ferme le flux (doit toujours être appelée quand on a fini d'écrire sur le flux).
- **flush()** : vide le flux de sortie et force l'écriture d'éventuels octets en mémoire tampon.

#### 1.2.3.2.2 Flux binaires de communication en sortie

Voici la liste des classes permettant l'écriture de données sous forme d'octets :

- **ByteArrayOutputStream** : implémente un flux de sortie dans lequel les données sont écrites dans un tableau de **byte**. Le buffer augmente au fur et à mesure que des données sont écrites. Il est possible de retrouver son contenu grâce aux méthodes **toString()** et **toByteArray()**.
- **FileOutputStream** : permet l'écriture de données dans un fichier. Il est possible de spécifier dans le constructeur si le contenu doit s'ajouter au contenu existant ou écraser les valeurs précédentes (si le fichier existait déjà). Par défaut, le contenu sera remplacé.
- **ObjectOutputStream** : cette classe permet la sérialisation des données.
- **PipedOutputStream** : travaille de paire avec un **PipedInputStream**.

#### 1.2.3.2.3 Flux binaires de traitement en sortie

Les classes suivantes héritent de **FilterOutputStream** (qui hérite elle-même de la classe **OutputStream**) et permettent de faire un traitement sur les données à écrire.

- **BufferedOutputStream** : cette classe contient un buffer associé au flux de sortie. En mettant en place un tel objet, cela permet d'écrire les données sans faire nécessairement d'appel système pour chaque **byte** à écrire.
- **DataOutputStream** : fournit des méthodes pour écrire des données sous formes de types primitifs Java, dans un flux de sortie, de manière portable. Ainsi, elles pourront être récupérées par la suite via un flux d'entrée.
- **PrintStream** : apporte au flux de sortie la capacité d'imprimer divers types de données (primitifs, **String** ou **Object** via appel à la méthode **toString()**).

### 1.2.4 Flux de caractères

Les flux de caractères, répartis en flux de caractères d'entrée et flux de caractères de sortie, sont pris en charge *via* un ensemble de classes :

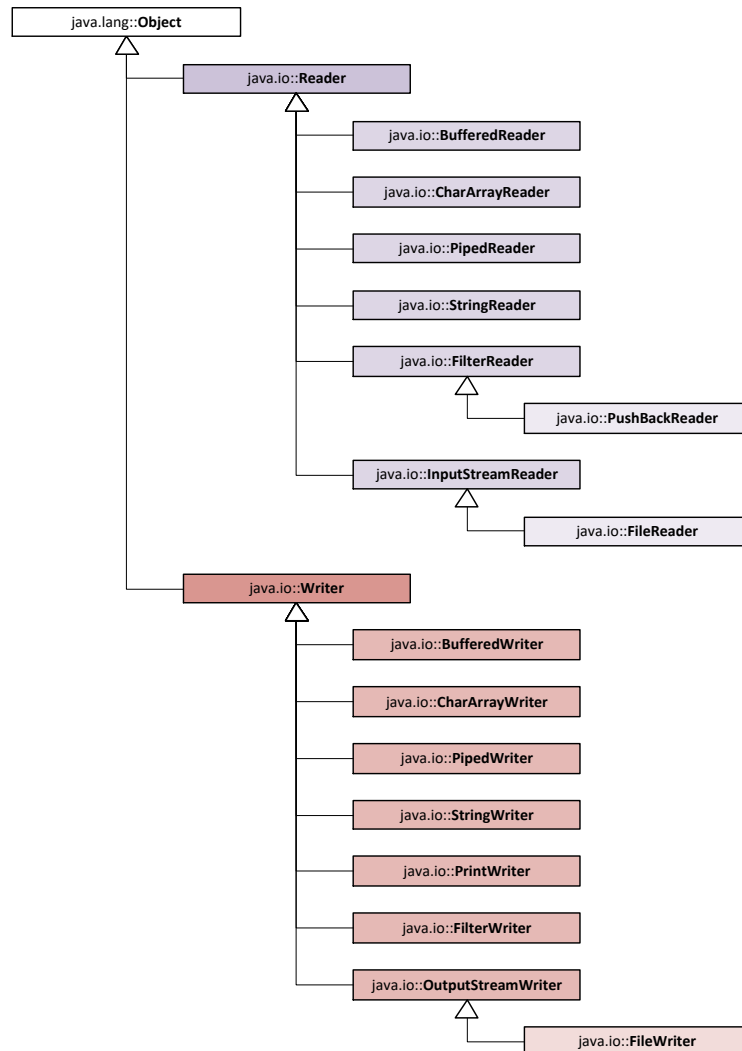


Figure 2 : classes prenant en charge les manipulations de flux de caractères

Comme pour les classes de flux binaires du paquetage, l'arborescence montre la séparation en deux grandes familles de classes : les « readers » (lecture) et les « writers » (écriture). Les classes héritant de la classe abstraite **Reader** permettent la lecture sur des flux de caractères. Les classes héritant de la classe **Writer**, quant à elles, servent pour l'écriture de flux de caractères. On remarque aussi la présence de classes permettant de faire un traitement sur le flux. Leurs super classes sont **FilterReader** et **FilterWriter**.



### 1.2.4.1 Flux de caractères en entrée

Les objets permettant la lecture de flux de caractères héritent tous de la classe abstraite **Reader**.

#### 1.2.4.1.1 La classe abstraite **Reader**

La classe **Reader** fournit des méthodes permettant la lecture de caractères. Ses méthodes **read()** sont similaires à celles des flux binaires. La seule méthode abstraite qu'elle implémente est la méthode **int read(char[] cbuff, int off, int len)** mais elle fournit aussi :

- **int read()** : lit le prochain caractère sur le flux et le retourne sous forme d'**int**. Pour l'utiliser réellement sous forme de caractère (**char**), il faudra convertir explicitement (cast).
- **int read(char[] cbuff)** : similaire à la méthode vue précédemment, elle stocke les caractères lus dans le tableau de **char**.
- **int read(char[] cbuff, int off, int len)** : similaire à la méthode vue précédemment, elle stocke les **len** caractères lus dans le tableau de **char** à partir de la position **off**.

La classe **Reader** possède également les méthodes **close()**, **mark(int readLimit)**, **markSupported()**, **reset()** et **skip(long n)**. Elle possède aussi une méthode **boolean ready()** qui teste s'il est possible de lire sans bloquer le flux.

#### 1.2.4.1.2 Flux de caractères pour la communication en entrée

Les flux suivants sont fournis dans le JDK :

- **FilterReader** : classe abstraite permettant de lire des flux de données filtrées. Elle possède des méthodes par défaut qui vont passer toutes les requêtes vers le flux qu'elle contient. Les sous-classes de **FilterReader** devront surcharger certaines de ces méthodes et pourront aussi fournir des méthodes et des champs supplémentaires.
- **PipedReader** : cette classe va être connectée à un **PipedWriter**, soit par la méthode **connect(PipedWriter p)** soit dès l'instanciation avec le constructeur **PipedReader(PipedWriter p)**.
- **InputStreamReader** : cette classe va permettre la communication entre les flux d'entrée d'octets et les flux d'entrée de caractères. En effet, il va lire les octets pour ensuite les transformer en caractères en utilisant un jeu de caractères spécifique (**Charset**). Ce jeu de caractères sera celui de la plateforme par défaut, mais peut être spécifié par l'utilisateur lors de l'instanciation avec **InputStreamReader(InputStream in, Charset cs)**.
- **StringReader** : cette classe est un flux de caractères créé à partir d'une chaîne de caractères. Cette chaîne de caractères va être fournie lors de l'instanciation à l'aide du constructeur **StringReader(String s)**.
- **CharArrayReader** : cette classe implémente une mémoire tampon (buffer) de caractères et peut être utilisée comme flux de caractères d'entrée.
- **BufferedReader** : cette classe va permettre de lire des flux de caractères d'entrée. Si la taille n'a pas été spécifiée lors du constructeur, elle sera assignée par défaut. Pour cela, deux constructeurs existent : le premier **BufferedReader(Reader in)** permet de créer un buffer à partir d'un flux d'entrée mais avec une taille par défaut tandis que le deuxième **BufferedReader(Reader in, int sz)** permet de spécifier également la taille du buffer.

### 1.2.4.2 Flux de caractères en sortie

Les objets permettant l'écriture sur des flux de caractères héritent tous de la classe abstraite **Writer**.

#### 1.2.4.2.1 La classe abstraite **Writer**

Cette classe abstraite permet l'écriture de flux de caractères. Elle impose l'implémentation des méthodes suivantes :

- **close()** : qui permet la fermeture d'un flux (il faudra tout d'abord vider les ressources utilisées par le flux avec la méthode **flush()**).
- **flush()** : permet justement de vider les ressources utilisées par le flux courant.
- **write(char[] cbuf, int off, int len)** : permet l'écriture d'intervalle de tableau de caractères.

#### 1.2.4.2.2 Flux de caractères pour la communication en sortie

Les flux suivants sont fournis dans le JDK :

- **FilterWriter** : classe abstraite permettant d'écrire des flux de données filtrées. Elle possède des méthodes par défaut qui vont passer toutes les requêtes vers le flux qu'elle contient. Les sous-classes de **FilterWriter** devront surcharger certaines de ces méthodes et pourront aussi fournir des méthodes et des champs supplémentaires.
- **PipedWriter** : cette classe va être connectée à un **PipedReader** soit par le constructeur **PipedWriter(PipedReader snk)** soit par la méthode **connect(PipedReader snk)**.
- **PrintWriter** : cette méthode implémente toutes les méthodes de la classe **PrintStream**. Les méthodes de la classe **PrintWriter** ne déclenchant pas d'exception, ce sera à l'utilisateur de gérer cela en utilisant la méthode **checkError()**.
- **BufferedWriter** : permet l'écriture de texte vers un flux de sortie de caractères, en mettant en mémoire tampon les caractères pour une écriture efficace de caractères, de tableaux et de chaînes de caractères.
- **OutputStreamWriter** : cette classe va permettre la communication entre les flux d'entrée de caractères et les flux d'entrée d'octets. En effet, il va écrire les caractères qu'il aura encodés en utilisant un jeu de caractères spécifique (**Charset**). Ce jeu de caractères sera celui de la plateforme par défaut, mais peut être spécifié par l'utilisateur lors de l'instanciation, avec **OutputStreamWriter(OutputStream out, Charset cs)**.
- **CharArrayWriter** : cette classe implémente une mémoire tampon de caractères qui peut être utilisée comme un objet **Writer**. On peut spécifier sa taille à l'aide du constructeur **CharArrayWriter**.
- **StringWriter** : cette classe représente un flux de caractères qui recueille sa sortie dans un objet **StringBuffer** lequel peut alors être utilisé pour construire une chaîne de caractères.

### 1.2.5 Utilisation des flux

Pour pouvoir obtenir le flux qui vous convient, il faut souvent passer par plusieurs constructions intermédiaires. Dans l'exemple suivant, on souhaite obtenir un flux sur un fichier, bufférisé et permettant de manipuler des données typées. Il est important de comprendre que, dans cet exemple, l'ordre de construction ne peut en aucun cas être changé. En effet, l'objet final à utiliser se doit d'être de type **DataInputStream**. Le second exemple montre comment écrire dans un tel flux.

#### Exemple

En entrée :

```
File f = new File("fichier.mp3");
FileInputStream fis = new FileInputStream(f);
BufferedInputStream bis = new BufferedInputStream(fis);
DataInputStream dis = new DataInputStream(bis);

int a = dis.readInt();
short s = dis.readShort();
boolean b = dis.readBoolean();
```

En sortie :

```
File f = new File("fichier.mp3");
FileOutputStream fos = new FileOutputStream(f);
BufferedOutputStream bos =
    new BufferedOutputStream(fos);
DataOutputStream dos = new DataOutputStream(bos);

int a = 10;        dos.writeInt(a);
short s = 3;       dos.writeShort(s);
boolean b = true;  dos.writeBoolean(b);
```

### 1.2.6 Exemple complet de manipulation des flux : copie d'un fichier

Ce programme attend 2 arguments sur la ligne de commande :

- Le nom du fichier source,
- Le nom du fichier cible.

Pour réaliser une copie de fichier, on travaille sur un flux binaire et non sur un flux textuel. On utilisera donc de préférence des classes dérivées de **InputStream** et de **OutputStream**. Il faut de plus penser à traiter les exceptions, notamment quand on manipule des flux.

## Exemple



```
import java.io.*;

public class Copy
{
    public static void main (String[] argv)
    {
        // Test sur le nombre de paramètres passés.
        if (argv.length != 2)
        {
            System.out.println("Usage > java Copy
                                sourceFile destinationFile");
            System.exit(0);
        }

        try
        {
            // Préparation du flux d'entrée.
            File sourceFile = new File(argv[0]);
            FileInputStream fis =
                new FileInputStream(sourceFile);
            BufferedInputStream bis =
                new BufferedInputStream(fis);
            long l = sourceFile.length();

            // Préparation du flux de sortie.
            FileOutputStream fos =
                new FileOutputStream(argv[1]);
            BufferedOutputStream bos =
                new BufferedOutputStream(fos);

            // Copie des octets du flux d'entrée
            // vers le flux de sortie.
            for (long i = 0; i < l; i++)
            { bos.write(bis.read()); }

            // Fermeture des flux de données.
            bos.flush();
            bos.close();
            bis.close();
        }
        catch (Exception e)
        {
            System.err.println("File access error !");
            e.printStackTrace();
        }

        System.out.println("Copie terminée");
    }
}
```

## 1.3 Compression

Les classes spécifiques à la compression ZIP sont issues du paquetage `java.util.zip`. Le JRE fournit également d'autres classes permettant de compresser dans d'autres formats (GZIP et JAR) dont les principes restent les mêmes.

### 1.3.1 Compression

La classe `ZipOutputStream` permet la compression des données *via* sa méthode `write(byte[] b, int off, int len)`. Elle est associée à un flux de communication (passé en paramètre au constructeur) sur lequel on lit les données compressées.

#### Exemple



```
try
{
    FileOutputStream fos =
        new FileOutputStream("C:\\dev\\archive.zip");
    ZipOutputStream zipos = new ZipOutputStream(fos);
    //...
    zipos.write(data, 0, count);
    // 'data' est un 'byte[]' contenant les données
    // à compresser et 'count' indique le nombre
    // de données de type 'byte' à écrire.
    //...
    zipos.close();
}
```

En Java, une archive ZIP peut être composée d'une ou plusieurs **ZipEntry**. Cette classe représente une entrée dans le fichier ZIP. En effet, une archive est généralement composée de plusieurs items correspondant à des fichiers (ou répertoire) compressés. Il est donc nécessaire de créer un **ZipEntry** pour chaque item. Son constructeur prend en paramètre un **String** correspondant au nom qu'aura l'item dans l'archive (généralement le nom du fichier). L'item est ensuite ajouté au **ZipOutputStream** grâce à la méthode `putNextEntry(ZipEntry e)`. Toutes les données écrites sur le flux seront alors associées à cet item jusqu'à l'appel de la méthode `closeEntry()`.

## Exemple

```
import java.io.*;
import java.util.zip.*;

public class Zip
{
    static final int BUFFER = 2048;

    public static void main(String[] args)
    {
        try
        {
            // On crée le fichier.
            FileOutputStream fos =
                new FileOutputStream("C:\\dev\\archive.zip");

            // On crée le flux de compression.
            ZipOutputStream zipos = new ZipOutputStream(fos);
            byte[] data = new byte[BUFFER];

            // On ouvre le flux sur le fichier à lire.
            FileInputStream fis =
                new FileInputStream("C:\\fichier.txt");

            // On crée un item correspondant au fichier
            // d'origine et on le place dans le ZIP.
            ZipEntry ze = new ZipEntry("fichier.txt");
            zipos.putNextEntry(ze);

            int count;
            // Tant qu'on peut lire...
            while ((count = fis.read(data, 0, BUFFER)) != -1)
            {
                // On écrit.
                zipos.write(data, 0, count);
            }

            // On ferme l'item.
            zipos.closeEntry();

            // On ferme le ZIP.
            zipos.close();
        }
        catch (Exception e)
        { e.printStackTrace(); }
    }
}
```



### 1.3.2 Décompression

La décompression s'effectue grâce à la méthode `read(byte[], int off, int len)` de la classe `ZipInputStream`. Cette méthode lit les données compressées d'un `ZipEntry` spécifique. Le `ZipEntry` doit donc être récupéré en premier *via* la méthode `getNextEntry()`. Pour savoir à quel fichier d'origine appartiennent les données, il suffit de faire appel à la méthode `getName()` du `ZipEntry`.

#### Exemple

```
import java.io.*;
import java.util.zip.*;

public class UnZip
{
    static final void main(String[] args)
    {
        try
        {
            // On crée le flux de communication en lecture.
            FileInputStream fis =
                new FileInputStream("C:\\dev\\archive.zip");

            // On crée le flux de décompression et on en
            // récupère chaque entrée.
            ZipInputStream zis = new ZipInputStream(fis);
            ZipEntry entry;
            while ((entry = zis.getNextEntry()) != null)
            {
                int count;
                byte[] data = new byte[BUFFER];

                // On récupère chaque item.
                FileOutputStream fos = new FileOutputStream
                    ("C:\\dezip\\" + entry.getName());
                // Tant que l'item contient des données
                // on les écrit.
                while ((count = zis.read(data, 0, BUFFER))
                    != -1)
                { fos.write(data, 0, count); }

                // On ferme le nouveau fichier.
                fos.close();
            }

            // On ferme l'archive.
            zis.close();
        }
        catch (Exception e)
        { e.printStackTrace(); }
    }
}
```



## 1.4 Sérialisation

Le besoin de persistance est un besoin :

- Très récurrent aux applications dès qu'elles manipulent un temps soit peu des données,
- Assez régulier dans ses fonctionnalités : enregistrer l'objet, charger l'objet, ...,
- Fastidieux à programmer (penser aux objets composés d'autres objets),
- Peu spectaculaire (c'est un attendu implicite du client).

On a cherché à automatiser cette tâche. Tous les langages évolués proposent des solutions pour gérer la persistance des objets et ces solutions sont souvent adossées à des BD (solutions intégrées dans les serveurs d'application actuels : Websphere - IBM, WebLogic - BEA, Oracle, .Net - MS, ...). En Java, le concept de persistance est fortement lié à celui de **sérialisation d'objet**.

### 1.4.1 Vue globale de la persistance en Java

Les objets sont, par défaut, éphémères. On dit qu'ils sont **transitoires**. En effet, dès qu'une application Java se termine, les objets sont détruits. Il est alors impossible de récupérer les données au redémarrage de l'application.

Il est donc nécessaire de pouvoir stocker les données sur une ressource externe à la mémoire de la JVM. Ceci peut être réalisé principalement de 2 façons :

- Via la sauvegarde sur un flux : la **sérialisation**,
- Via la sauvegarde au sein d'une BD, avec JDO par exemple.

Lorsque des objets sont sérialisés ou sauvegardés dans une BD, ils deviennent alors **persistants**. En effet, ils résident au-delà de la vie de l'application. Une fois sauvegardés (physiquement), ils peuvent être récupérés/reconstruits à un autre moment. Les données stockées sont les variables propres aux objets. Ce n'est donc pas vraiment l'objet en lui-même qui est sauvegardé (les méthodes ne sont pas conservées par exemple) mais son **état** interne (ses variables). Lors de la reconstruction de l'objet, il est nécessaire de posséder la classe (le *bytecode*) de l'objet à reconstruire.



#### Définition

**Persistance** : capacité d'une application à enregistrer les instances.



#### Exemple

Un compte bancaire doit persister avec ses nouvelles données après son utilisation pour une opération de transfert d'argents.

Il faut sauvegarder l'objet dans un fichier (que ce soit un fichier utilisateur ou *via* un SGBD). La question du SGBD relationnel ou objet pourrait se poser si la prédominance des SGBDR n'imposait pas ce type d'organisation des données.



### 1.4.2 Mécanisme de sérialisation en Java

C'est un mécanisme Java pour générer un flux à partir d'un objet : le mettre en « série d'octets ». Ce mécanisme est aussi bien valable pour un flux de type **File** que pour une **Socket** :

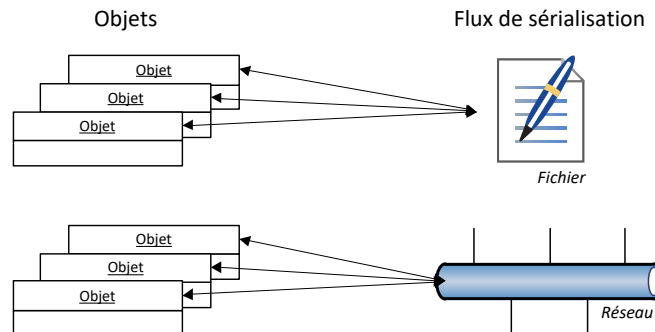


Figure 3 : mécanismes de sérialisation dans un flux

Le cas de la sauvegarde dans une BD est un peu plus complexe : faire persister les objets dans une BD implique de faire un *mapping* entre des objets et leurs représentation en relationnel (facture, compte bancaire, etc).

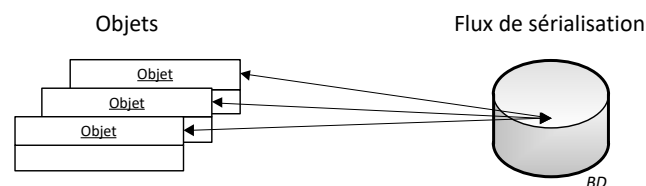


Figure 4 : mécanisme de sérialisation dans une BD relationnelle

La sérialisation permet d'enregistrer des arborescences d'objets interconnectés et de recharger ces objets ultérieurement. Pour bénéficier de ce mécanisme, un objet doit implémenter l'interface `java.io.Serializable`, sinon on lève l'exception `java.io.NotSerializableException`.

Il faut noter que plusieurs classes de l'API Java implémentent déjà l'interface **Serializable** : **String**, **Integer**, ...

#### 1.4.2.1 Interface et flux de sérialisation

L'interface **Serializable** s'utilise de façon particulière :

- Elle ne définit aucun champ ni aucune méthode,
- Par contre, il est possible d'appeler sur les instances des classes qui implémentent cette interface 2 méthodes particulières (**writeObject()** pour la sérialisation et **readObject()** pour la désérialisation) sans avoir à les implémenter,
- Ces 2 méthodes possèdent un comportement par défaut qui, si nécessaire, pourra être modifié en les redéfinissant dans la classe concernée,
- Il en va de même avec 2 autres méthodes particulières : **writeReplace()** et **readResolve()**.

La sérialisation/désérialisation d'un objet utilise une paire d'interfaces spéciales : **ObjectOutput** et **ObjectInput**. Ces interfaces sont implémentées en fonction du type de flux désiré en sortie, par exemple **java.io.ObjectOutputStream** et **java.io.ObjectInputStream** (qu'on peut combiner avec d'autres flux qui écrivent ou lisent dans des fichiers, des réseaux ou d'autres sources pour stocker ou transmettre les objets).

#### 1.4.2.2 Problèmes dus à la sérialisation

Les attributs d'un objet ne sont pas tous de type scalaire (**int**, **float**, **double**, ...). Certains sont des agrégats (objets ou tableaux) : ce sont donc des références. On ne va pas sérialiser un pointeur sur une zone mémoire qui, certes, contient les informations à un instant donné mais risque d'être libérée à tout moment : c'est l'objet pointé qu'il faut sauvegarder. Faut-il aussi déterminer si un attribut est de type scalaire ou un agrégat et adapter le comportement de la sérialisation ? Comment sérialiser deux objets qui se pointent mutuellement (risque de boucle infinie, ce qui reviendrait à sauvegarder les mêmes objets indéfiniment) ?

Le flux de sortie **ObjectOutputStream** possède un certain nombre de mécanismes permettant de résoudre ces problèmes :

- Il mémorise chaque objet qui lui est transmis pour écriture et n'écrit pas deux fois le même (la seconde tentative d'écriture produit une référence vers la première copie de l'objet qui est en train d'être écrit). Ainsi tout objet n'est sauvé qu'une fois grâce à un mécanisme de cache (il est donc possible de sauver une liste circulaire, à condition que tous ses éléments soient sérialisables).
- La méthode **reset()** demande au flux **ObjectOutputStream** d'effacer (« d'oublier ») tous les objets qu'il a écrits auparavant.
- Lorsqu'un objet est sauvé, tous les objets qui peuvent être atteints depuis cet objet sont également sauvés (donc, en particulier si l'on sauve le premier élément d'une liste, tous les éléments de la liste sont sauvés, ici aussi à condition bien sûr que tous ses éléments soient sérialisables).

D'autre part, il est à noter que :

- En implémentant **Serializable**, un objet peut être enregistré ou récupéré même si une version différente (mais compatible) de sa classe est présente,
- Le comportement par défaut de la sérialisation sauvegarde dans le flux tous les champs qui ne sont ni **static** ni **transient**.
- Des informations sur la classe (nom, version) ainsi que le type et le nom des champs de l'instance sérialisée sont également sauvegardés afin de permettre la récupération future de l'objet (la sérialisation utilise la réflexivité Java pour identifier dans l'objet les données membres à sauvegarder),
- Quelques classes ne sont pas sérialisables : **Thread**, **InputStream**, **OutputStream**, **Peer**, **JDBC**, ...

### 1.4.2.3 Sérialisation et héritage

Si une classe sérialisable étend une classe qui ne l'est pas :

- Les valeurs des données membres de l'objet parent ne seront pas écrites sur le flux de sortie **ObjectOutputStream**,
- De plus, lors de la reconstruction de l'objet, le constructeur sans paramètre sera appelé pour l'initialisation des données membres héritées de l'objet parent non sérialisable.

Dans le cas d'une classe sérialisable étendant une classe elle-même sérialisable, la sérialisation se fait de façon classique (*via* celle de la classe et de sa super classe, ...).

Les sous-classes d'une classe sérialisable sont elles-mêmes sérialisables.

### 1.4.2.4 Mise en œuvre

Pour créer une classe afin qu'elle soit sérialisable, il suffit simplement de faire implémenter par cette classe l'interface **java.io.Serializable**.

#### Exemple



```
import java.io.*;
import java.util.*;

public class MaClasse implements Serializable
{
    private String monPremierAttribut;
    private Date monSecondAttribut;
    private long monTroisiemeAttribut;

    // D'éventuelles méthodes
}
```

Il est parfois nécessaire de modifier quelque peu le mécanisme de sérialisation par défaut, par exemple si l'on ne souhaite pas qu'un champ soit enregistré (dans le cas d'un champ spécifique à une session par exemple). On utilise un indicateur d'usage particulier pour indiquer qu'une donnée membre est **transitoire** : le mot-clé **transient**.

#### Exemple



```
import java.io.*;
import java.util.*;

public class MaClasse implements Serializable
{
    private String monPremierAttribut;
    private transient Date monSecondAttribut;
    private long monTroisiemeAttribut;

    // D'éventuelles méthodes
}
```

En fait, une donnée membre mentionnée comme transitoire (**transient**) est bien sauvegardée lors de la sérialisation mais pas sa valeur : seuls son type et son nom sont conservés. Lors de la désérialisation, une donnée membre transitoire est donc bien restaurée et la valeur qui lui est affectée est la valeur par défaut de son type :

- 0 pour les types numériques,
- **false** pour le type **boolean**,
- **null** pour les objets.

Si les besoins sont plus spécifiques et ne peuvent être traités en n'utilisant que l'indicateur d'usage **transient**, on peut redéfinir les méthodes **readObject()** et **writeObject()** qui sont appelées pour le chargement et la sauvegarde des objets. Ces deux méthodes devront être redéfinies dans les classes pour les instances desquelles on souhaite mettre en œuvre ces traitements particuliers lors de la sérialisation/désérialisation :



#### Données

```
private void writeObject(java.io.ObjectOutputStream out)
    throws IOException
```

```
private void readObject(java.io.ObjectInputStream in)
    throws IOException, ClassNotFoundException
```

### 1.4.2.5 Exemples de sérialisation

#### 1.4.2.5.1 Exemple de classe persistante



#### Exemple (début)

```
import java.io.Serializable;

public class Animal implements Serializable
{
    int age;
    boolean vaccin;
    String couleur;

    Animal()
    {
        this.age = 0;
        this.vaccin = false;
        this.couleur = "";
        System.out.println("Un animal a été créé.");
    }
}
```



#### Exemple (fin)

```
Animal(int a, boolean v, String c)
{
    this.age = a;
    this.vaccin = v;
    this.couleur = c;
    System.out.println("Un animal a été créé.");
}
```

#### 1.4.2.5.2 Exemple de classe sérialisant une classe persistante



#### Exemple

```
import java.io.*;

public class Serial
{
    public static void main(String[] args)
        throws IOException
    {
        Animal chien = new Animal(5, true, "noir");
        FileOutputStream fos =
            new FileOutputStream("SaveAnimal.txt");
        ObjectOutputStream oos =
            new ObjectOutputStream(fos);

        oos.writeObject(chien);
        oos.close();
    }
}
```

#### 1.4.2.5.3 Exemple de classe désérialisant une classe persistante



#### Exemple (début)

```
import java.io.*;

public class Deserial
{
    public static void main(String[] args)
        throws IOException, ClassNotFoundException
    {
        FileInputStream fis = new FileInputStream(args[0]);
        ObjectInputStream ois = new ObjectInputStream(fis);
```



### Exemple (fin)

```
Animal monObjetDeserialise =
    (Animal)ois.readObject();
ois.close();

System.out.println("Age de l'objet désérialisé : "
    + monObjetDeserialise.age);
System.out.println("Couleur : "
    + monObjetDeserialise.couleur);
if (monObjetDeserialise.vaccin)
    System.out.println("L'objet est vacciné !");
}
```

#### 1.4.2.5.4 Exemple complet de sérialisation/désérialisation



### Exemple (début)

```
import java.io.*;
import java.awt.*;
import javax.swing.*;

public class Serialisation
{
    private final static Reader reader
        = new InputStreamReader(System.in);
    private final static BufferedReader keyboard
        = new BufferedReader(reader);

    // Permet de créer une fenêtre et de
    // la sérialiser dans un fichier.
    public void saveWindow() throws IOException
    {
        JFrame window = new JFrame("Ma fenêtre");
        JPanel pane = (JPanel>window.getContentPane();

        pane.add(new JLabel("Barre de status"),
            BorderLayout.SOUTH);
        pane.add(new JTree(), BorderLayout.WEST);
        JTextArea textArea =
            new JTextArea("Ceci est le contenu !!!");
        textArea.setBackground(Color.GRAY);
        pane.add(textArea, BorderLayout.CENTER);
    }
}
```

### Exemple (suite)



```
JPanel toolbar = new JPanel(new FlowLayout());
toolbar.add(new JButton("Open"));
toolbar.add(new JButton("Save"));
toolbar.add(new JButton("Cut"));
toolbar.add(new JButton("Copy"));
toolbar.add(new JButton("Paste"));

pane.add(toolbar, BorderLayout.NORTH);
window.setSize(400,300);

FileOutputStream fos =
    new FileOutputStream("window.ser");
ObjectOutputStream oos =
    new ObjectOutputStream(fos);
oos.writeObject(window);
oos.flush();
oos.close();
}

// Permet de reconstruire la fenêtre
// à partir des données du fichier.
public void loadWindow() throws Exception
{
    FileInputStream fis =
        new FileInputStream("window.ser");
    ObjectInputStream ois =
        new ObjectInputStream(fis);
    JFrame window = (JFrame)ois.readObject();
    ois.close();

    window.setVisible(true);
}

// Permet de saisir différentes commandes.
// Testez plusieurs load consécutifs : plusieurs
// fenêtres doivent apparaître.
public static void main(String[] args)
    throws Exception
{
    Serialisation object = new Serialisation();

    while(true)
    {
        System.out.print("Saisir le mode d'exécution
        (load ou save) : ");
        String mode = keyboard.readLine();
```

**Exemple (fin)**

```
if (mode.equalsIgnoreCase("exit"))
    break;
if (mode.equalsIgnoreCase("save"))
    object.saveWindow();
if (mode.equalsIgnoreCase("load"))
    object.loadWindow();
}

System.exit(0);
}
```

**1.4.2.6 Une précaution à prendre**

Nous avons vu que le flux **ObjectOutputStream** permettait de ne sérialiser chaque objet qu'une seule fois. Ceci est réalisé *via* l'utilisation d'un vecteur (**Vector**) d'objets : à chaque fois qu'un objet est sérialisé sur le flux, sa référence est rajoutée au vecteur. Lorsque l'on souhaite écrire un nouvel objet sur le flux, on vérifie préalablement dans le vecteur si sa référence existe ou non. Si elle existe, c'est qu'il a déjà été sérialisé.

Ce mode de fonctionnement peut poser un problème, notamment si on doit sérialiser beaucoup d'objets. En effet, dans ce cas il y a des risques que le vecteur associé aux flux de sérialisation « grossisse » jusqu'à ne plus tenir en mémoire (on « obtient » alors une erreur **java.lang.OutOfMemoryError**). On a la possibilité de pouvoir réinitialiser le flux (et donc le vecteur associé) pendant une opération de sérialisation grâce à la méthode **reset()** du flux. Ceci peut donc éviter le problème de surcharge de la mémoire mais cette solution ne peut être utilisée que si on est certains que les objets à sérialiser après l'appel de la méthode **reset()** n'ont pas déjà été sauvegardés dans le flux. Si tel n'est pas le cas, certaines instances pourraient éventuellement être sérialisées 2 fois.



## 2 Manipulation des flux sur sockets : communications réseau

Les paquetages `java.net` et `javax.net` ainsi que les classes qu'ils fournissent font de Java un très bon langage de programmation réseau. Les classes du paquetage `java.net` encapsulent le concept des « sockets » du projet Berkeley Software Distribution (BSD), inventé à l'université de Berkeley en Californie.

### 2.1 Concept de socket

Un **socket** est un point de terminaison dans une communication bidirectionnelle entre deux programmes fonctionnant sur un réseau. Un socket est associé à un numéro de port afin que la couche TCP puisse identifier l'application vers laquelle les données doivent être transmises. En fonctionnement normal, une application serveur fonctionne sur un ordinateur et possède un socket d'écoute associé à un port d'écoute. Le serveur attend une demande de connexion de la part d'un client sur ce port.

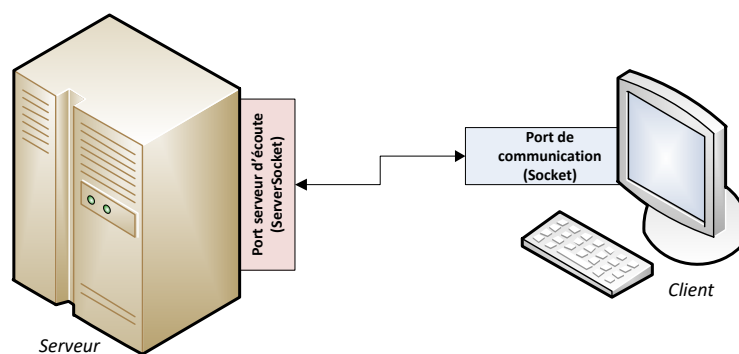


Figure 5 : demande de connexion en mode TCP d'un client à un serveur

Si tout se passe bien, le serveur accepte la connexion. Suite à cette acceptation, le serveur crée un nouveau socket associé à un nouveau port. Ainsi il pourra communiquer avec le client, tout en continuant l'écoute sur le socket initial en vue d'autres connexions.

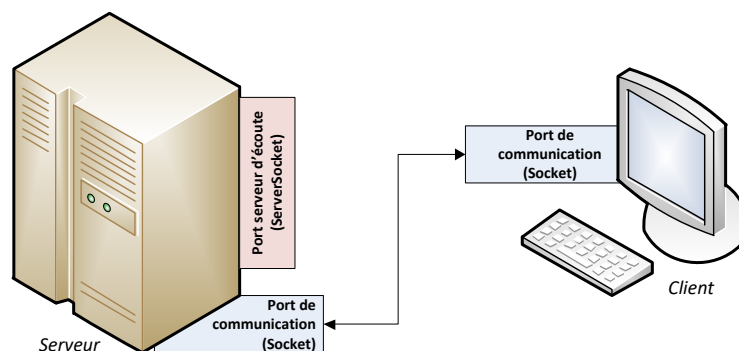


Figure 6 : suite de la communication en mode TCP entre le client et le serveur

## 2.2 Les paquetages de Java dédiés

Les paquetages `java.net` et `javax.net` fournissent les classes et interfaces suivantes :

- Adresses IP : **InetAddress**,
- Socket TCP : **Socket**, **ServerSocket**, **JSSE** (Java Secure Socket Layer),
- Socket UDP : **DatagramSocket**, **DatagramPacket**,
- Socket MultiCast : **MulticastSocket**, **DatagramPacket**,
- Classes niveau application (Couche 7 OSI) : **URL**, **URLConnection**, **HttpURLConnection**, **JarURLConnection**.

Dans ce cours nous ne nous intéresserons pas à **JSSE**, aux **MulticastSocket** et aux **URL**.

## 2.3 La classe `java.net.InetAddress`

La classe **InetAddress** représente une adresse du protocole IP, aussi bien par son adresse IP que par son nom d'hôte. Cette classe peut être vue comme la représentation d'une adresse IP et est utilisée par les classes **Socket** et **DatagramSocket**.

Les méthodes suivantes permettent la résolution DNS :

- Obtenir l'adresse ou les adresses de l'hôte dont le nom est passé en paramètre :



### Données

```
public static InetAddress getByName(String hostname)
    throws UnknownHostException
```

```
public static InetAddress[] getAllByName(String
    hostname) throws UnknownHostException
```

- Obtenir l'adresse locale :



### Données

```
public static InetAddress getLocalHost()
```

## 2.4 Utilisation des sockets en mode connecté (TCP)

Les sockets TCP/IP sont utilisés pour établir des échanges de flux fiables, bidirectionnels et point à point entre différentes machines sur Internet. Les sockets peuvent également permettre à une application Java de communiquer avec une tout autre application de la machine locale où n'importe où sur Internet. En mode connecté (TCP), une connexion fiable est établie entre client et serveur (contrôle d'erreur, renvoi de paquets...). Java considère deux types de sockets TCP :

- La classe **ServerSocket** permet d'implémenter le socket d'écoute serveur. Celui-ci attend les demandes de connexion et les accepte.
- La classe **Socket** permet d'implémenter un socket de communication. C'est donc au sein de cette classe que l'on trouve les méthodes de communication.

### 2.4.1 Du côté du serveur (`java.net.ServerSocket`)

Sur le serveur, le processus d'exécution d'une communication par socket en mode connecté est le suivant :

1. Création d'un **ServerSocket**,
2. Attente d'une demande de connexion et création d'une socket dédiée sur le serveur lorsqu'une connexion est acceptée : **accept()**,
3. Récupération des flots d'entrée et de sortie de la socket dédiée précédemment créée : **getInputStream()** et **getOutputStream()**,
4. Échange de données avec le client *via* la socket dédiée : **read()**, **readLine()**, **write()**, **println()**, **flush()**,
5. Fermeture de la connexion *via* la socket dédiée : **close()**,

Pendant les étapes 3, 4 et 5, le **ServerSocket** reste disponible pour attendre des demandes de connexion d'un autre client : **accept()**.

#### *Création d'une socket d'écoute :*



##### Exemple

```
ServerSocket serveur = new ServerSocket(port);
```

#### *Attente des demandes et création du socket de communication :*



##### Exemple

```
Socket communication = serveur.accept();  
// Bloquant en attente de connexion
```

#### *Constructeurs de la classe `ServerSocket` :*



##### Données

```
public ServerSocket(int port) throws IOException  
// Crée une socket à l'écoute du port spécifié.
```

```
public ServerSocket(int port, int backlog)  
    throws IOException  
// Crée une socket à l'écoute du port spécifié en  
// spécifiant la taille de la file d'attente  
// des demandes de connexion.
```

```
public ServerSocket(int port, int backlog,  
    InetAddress binAddress) throws IOException  
// Comme le constructeur précédent mais en restreignant  
// l'adresse sur laquelle on accepte les connexions.
```

### Principales méthodes de la classe *ServerSocket* :



#### Données

```
public Socket accept() throws IOException
// Accepte la connexion et renvoie un nouveau Socket
```

```
public void setToTimeout(int timeout)
    throws SocketException
// Cette méthode prend en paramètre le délai de
// garde exprimé en millisecondes. La valeur
// par défaut, 0, équivaut à l'infini. À
// l'expiration du délai de garde, l'exception
// java.io.InterruptedIOException est levée.
```

```
public void close()
// Ferme la socket d'écoute.
```

```
public InetAddress getInetAddress()
// Retourne l'adresse à partir de laquelle
// la socket écoute.
```

```
public int getLocalPort()
// Retourne le port sur lequel la socket écoute.
```

### 2.4.2 Du côté du client (*java.net.Socket*)

Sur le client, le processus d'exécution d'une communication par socket en mode connecté est le suivant :

1. Création d'une **Socket**,
2. Récupération des flots d'entrée et de sortie : **getInputStream()** et **getOutputStream()**,
3. Échange de données avec le serveur : **read()**, **readLine()**, **write()**, **println()**, **flush()**,
4. Fermeture de la connexion : **close()**.

#### Création de la socket cliente :



#### Exemple

```
Socket Client = new Socket(host, port);
```

#### Communication (utilisation des flux *InputStream*, *OutputStream* et de leurs dérivés) :



#### Exemple

```
InputStream entree = Client.getInputStream();
OutputStream sortie = Client.getOutputStream();
```

### Constructeurs de la classe *Socket* :



#### Données

```
public Socket()
// Crée une socket non-connectée.
```

```
public Socket(InetAddress address, int port)
    throws IOException
// Crée une socket de communication et établit une
// connexion sur le port voulu de la machine
// dont l'adresse IP est spécifiée.
```

```
public Socket (String host, int port)
    throws UnknownHostException, IOException
// Crée une socket de communication et établit une
// connexion sur le port voulu de la machine dont le
// nom d'hôte est spécifié.
```

```
public Socket(InetAddress address, int port
    InetAddress localAddress, int localPort)
    throws IOException
// Crée une socket de communication et établit une
// connexion sur le port voulu de la machine
// dont l'adresse IP est spécifiée.
// Spécifie en plus le port et l'adresse IP du client.
```

```
public Socket(String host, int port,
    InetAddress localAddress, int localPort)
    throws IOException
// Crée une socket de communication et établit une
// connexion sur le port voulu de la machine dont le
// nom d'hôte est spécifié.
// Spécifie en plus le port et l'adresse IP du client.
```

### Principales méthodes de la classe *Socket* :

- La communication effective sur une socket est basée sur les flux de données `java.io.InputStream` et `java.io.OutputStream` :



#### Données

```
public InputStream getInputStream() throws IOException
// Retourne le flux d'entrée de la socket.
```

```
public OutputStream getOutputStream() throws IOException
// Retourne le flux de sortie de la socket.
```

- Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer un délai de garde pour l'attente des données (similaire au délai de garde de la socket d'écoute) :



#### Données

```
public void setTimeout(int timeout)
    throws SocketException
```

- Un ensemble de méthodes permet d'obtenir les éléments constitutifs de la liaison établie :



#### Données

```
InetAddress getAddress()
// Retourne l'adresse à laquelle la socket est
// connectée. L'adresse de type InetAddress concatène
// le nom de domaine et l'adresse IP correspondante
// (par exemple : www.yahoo.com/216.109.117.207).
```

```
int getPort()
// Retourne le port distant sur lequel la socket
// est connectée.
```

```
InetAddress getLocalAddress()
// Retourne l'adresse locale à laquelle la socket
// est associée.
```

```
int getLocalPort()
// Retourne le port local associé à la socket.
```

- Fermeture de la socket (et libération des ressources associées) :



#### Données

```
void close()
```

### 2.4.3 Flux utilisés

La communication effective sur une connexion par socket est basée sur les flux binaires de données (`java.io.OutputStream` et `java.io.InputStream`). Une opération de lecture sur ces flots est bloquante tant que des données ne sont pas disponibles. Cependant, il est possible de fixer un délai de garde pour l'attente des données grâce à la méthode `setTimeout` (similaire au délai de garde de la socket d'écoute : levée de l'exception `java.io.InterruptedIOException`).

## 2.4.4 Exemple complet de client/serveur en mode connecté (TCP)

### 2.4.4.1 Du côté du serveur

#### Exemple

```
// Exemple simple : le serveur accepte une connexion,
// reçoit un entier envoyé depuis le client et renvoie
// au client le même entier incrémenté de 1.
import java.net.*;
import java.io.*;

public class ServeurBasique
{
    public static void main(String[] args)
    {
        try
        {
            ServerSocket ecoute = new ServerSocket(18000, 5);
            Socket service = (Socket) null;

            while (true)
            {
                service = ecoute.accept();
                OutputStream os = service.getOutputStream();
                InputStream is = service.getInputStream();

                os.write(is.read() + 1);
                service.close();
            }
        }
        catch (Exception e)
        { /* ... */ }
    }
}
```



### 2.4.4.2 Du côté du client

#### Exemple



```
// Exemple simple : le serveur accepte une connexion,  
// reçoit un entier envoyé depuis le client et renvoie  
// au client le même entier incrémenté de 1.  
import java.net.*;  
import java.io.*;  
  
public class ClientBasique  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            Socket s = new Socket("localhost", 18000);  
            OutputStream os = s.getOutputStream();  
            InputStream is = s.getInputStream();  
  
            os.write((int) 'a');  
            System.out.println((char) is.read());  
            s.close();  
        }  
        catch (Exception e)  
        { /* ... */ }  
    }  
}
```

### 2.4.5 Les exceptions

Les principales exceptions à considérer sont :

- **java.net.BindException** : erreur de liaison à une adresse locale liée,
- **java.net.ConnectException** : refus de connexion par l'hôte,
- **java.net.NoRouteToHostException** : hôte injoignable (route non trouvée),
- **java.net.ProtocolException** : erreur de protocole (TCP, ...),
- **java.net.SocketException** : erreur de protocole (TCP, ...),
- **java.net.UnknownHostException** : erreur de DNS.



## 2.5 Sockets en mode datagramme (UDP)

Le protocole TCP/IP répond à la majorité des besoins réseaux. Il permet de transporter de manière très fiable des paquets de données. Toutefois, du fait de l'exécution de nombreux algorithmes de traitement, des congestions du réseau ou des paquets perdus, il constitue une solution de transport coûteuse.

Les **datagrammes** représentent une solution alternative. Un datagramme est un ensemble d'informations transmissibles de machine en machine. Aucun contrôle n'existe pour déterminer si un datagramme est bien arrivé à destination ou s'il a été intercepté par une autre machine. De même, il n'existe aucun moyen permettant de savoir s'il a été endommagé au cours du transfert.

Java implémente les datagrammes dans le cadre du protocole UDP et fournit 2 classes :

- **DatagramPacket** : permet de créer des objets contenant les données du datagramme. Un **DatagramPacket** possède une zone de données, un numéro de port et éventuellement une adresse IP.
- **DatagramSocket** : mécanisme qui permet d'envoyer et de recevoir des **DatagramPacket**.

**Constructeurs :**



Données

```
public DatagramSocket() throws SocketException  
// Construit une socket datagramme.
```

```
public DatagramSocket(int port) throws SocketException  
// Construit une socket datagramme en spécifiant  
// un port sur la machine locale.
```

```
public DatagramPacket(byte[] buf, int length)  
// Construit un paquet en mode datagramme.
```

```
public DatagramPacket(byte[] buf, int length,  
    InetAddress address, int port)  
// Construit un paquet en mode datagramme en  
// spécifiant l'adresse et le port de destination.
```

**Émission/réception :**



Données

```
public void send(DatagramPacket p) throws IOException  
// Envoi d'un paquet.
```

```
public void receive(DatagramPacket p) throws IOException  
// Réception d'un paquet.
```

**Connexion** : il est possible de « connecter » un socket en mode datagramme à un destinataire. Dans ce cas, les paquets émis sur le socket le seront toujours pour l'adresse spécifiée. La connexion simplifie l'envoi d'une série de paquets (il n'est plus nécessaire de spécifier l'adresse de destination pour chacun d'entre eux) et accélère les contrôles de sécurité (ils ont lieu une fois pour toutes à la connexion).



#### Données

```
public void connect(InetAddress adress, int port)
// Connexion.
```

**Déconnexion** : enlève l'association (la socket redevient disponible comme dans son état initial).



#### Données

```
public void disconnect()
// Déconnexion.
```

#### Autres méthodes utiles :



#### Données

```
public InetAddress getAddress()
// Retourne l'adresse de destination d'un paquet sous
// la forme d'un objet InetAddress.
```

```
public int getPort()
// Retourne le numéro du port de destination.
```

```
public byte[] getData()
//Retourne le tableau de données fourni par
// un datagramme. On utilise souvent cette méthode
// à la réception d'un DatagramPacket.
```

```
public int getLength()
// Retourne la taille des données valides continues
// dans le tableau de données du datagramme.
```

```
public void close()
//Libère les ressources du système associées
// à la socket.
```

**Exemple complet d'utilisation :**

- Pour l'envoi de datagrammes :

**Exemple**

```
import java.net.*;
import java.io.*;

public class EnvoiMessage
{
    // Port sur lequel on envoie les données.
    static final int port = 47;

    public static void main(String[] argv)
    {
        // On vérifie si l'adresse IP ou le nom de
        // la machine destinataire est spécifié/
        if (argv.length != 1)
        {
            System.err.println("Donnez le nom de
            la machine destinataire");
            System.exit(0);
        }

        BufferedReader entree = new BufferedReader(
            new InputStreamReader(System.in));
        InetAddress adresse = null;
        DatagramSocket socket;

        // Création de l'adresse du destinataire.
        try
        {
            adresse = InetAddress.getByName(argv[0]);

            // Envoi du message.
            String ligne = "Hello world";
            byte[] message = new byte[ligne.length()];
            message = ligne.getBytes();
            DatagramPacket envoi =
                new DatagramPacket(message,
                    ligne.length(), adresse, port);
            socket = new DatagramSocket(port);
            socket.send(envoi);
        }
        catch (Exception e)
        { System.err.println("Erreur d'envoi."); }
    }
}
```



- Pour la réception de datagrammes :

#### Exemple

```
import java.io.*;
import java.net.*;

class ReceptionMessage
{
    // Port de reception.
    static final int port = 47;

    public static void main(String[] argv)
    {
        byte[] receptionOctets = new byte[1000];
        String texte;

        try
        {
            DatagramSocket socket = new DatagramSocket(port);
            DatagramPacket reception =
                new DatagramPacket(receptionOctets,
                    receptionOctets.length);

            socket.receive(reception);
            texte = new String(receptionOctets);

            System.out.println("Réception de " +
                reception.getAddress().getHostName());
            System.out.println("Sur le port " +
                reception.getPort());
            System.out.println("Texte : " + texte);
        }
        catch (Exception e)
        {
            System.err.println("Erreur lors de
                l'ouverture de la socket.");
        }
    }
}
```



## 3 Gestion programmatique des classes

La gestion programmatique des classes repose sur l'introspection, la réflexivité. Ces concepts offrent :

- La possibilité pour une classe de se décrire,
- La possibilité pour une classe d'obtenir des informations sur une autre classe.

La description d'une classe comprend :

- Ses données membres (type, modificateur(s), visibilité),
- Ses fonctions membres (paramètre(s), type de retour, modificateur(s), visibilité),
- Ses constructeurs,
- Les exceptions levées,
- Le paquetage d'appartenance,
- D'autres informations complémentaires (super classe, interfaces implémentées, ...).

### 3.1 Réification

En Java, la réification se définit au travers des propriétés suivantes :

- Une classe est un objet, on dispose donc d'une classe **Class**.
- Une méthode est un objet, on dispose donc d'une classe **Method**.
- Un champ est un objet, on dispose donc d'une classe **Field**.
- ...

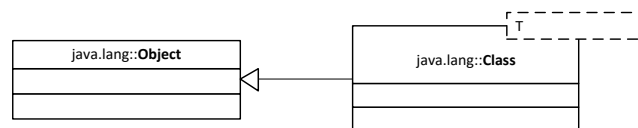


Figure 7 : principe de la réification

### 3.2 Intérêt et utilisation

L'intérêt est de pouvoir instancier des classes de manière dynamique et de pouvoir affecter leurs champs et appeler leurs méthodes.

La différence entre l'instanciation « classique » (statique) et l'instanciation dynamique est à noter :



#### Exemple

Instanciation classique :

```
CompteBancaire cb = new CompteBancaire();
```



#### Exemple

Instanciation dynamique :

```
// En supposant que args[0] = "CompteBancaire".
Object o = Class.forName(args[0]).newInstance();
```

Dans quel cadre l'utiliser ?

- Les debuggers,
- Les plateformes J2EE gérant le cycle de vie d'objets,
- Les environnements de développement (pour la complétion par exemple),
- Tomcat avec les servlets (Tomcat est un conteneur de pages JSP qui, entre autres fonctionnalités, transforme une page JSP en une servlet, peut pré-lancer des objets, les charger à la demande, ... ; il a donc besoin de tout savoir sur ces objets),
- Les environnements qui permettent de **sauvegarder les objets** dans des **BD relationnelles** avec une prise en charge minime de la part du développeur : on peut imaginer qu'un programme détecte les données membres d'une instance, cherche la table dans la BD qui dispose de ces informations comme champs et sauvegarde l'objet ou crée la table adéquate si elle n'existe pas.

### 3.3 Outils de mise en œuvre

Trois outils sont importants dans la mise en œuvre des mécanismes d'introspection :

- Le paquetage `java.lang.reflect`, qui contient par exemple les classes `Field` et `Method` et, en particulier, la méthode `invoke()` qui permet d'invoquer dynamiquement une méthode sur un objet.
- La classe `java.lang.Object`, notamment avec la méthode `public final Class getClass()` qui retourne la classe de l'objet sur lequel elle est utilisée (un objet de la classe `Class`),
- La classe `java.lang.Class`, notamment avec les méthodes :
  - `public Class[] getInterfaces()` qui retourne les éventuelles interfaces implémentées par cette classe,
  - `public Constructor[] getConstructors() throws SecurityException` qui retourne un tableau des constructeurs publics de la classe,
  - `public String getName()` qui renvoie le nom d'une classe,
  - `public Field[] getDeclaredFields() throws SecurityException` qui retourne un tableau des champs non hérités,
  - `public Method[] getDeclaredMethods() throws SecurityException` qui retourne un tableau des méthodes non héritées.

### 3.4 Chargement dynamique des classes

En Java, on peut charger (linker) des classes « compilées » en utilisant leur nom au moment de l'exécution (elles ne sont donc pas connues à la compilation).

**Objectif :** étendre les capacités d'un programme en cours d'exécution (similaire aux DLL Windows).

Pour charger dynamiquement une classe, on dispose de deux solutions :

- Via la classe `java.lang.ClassLoader`,
- Via la classe `Class`.

### 3.4.1 Première solution de chargement dynamique de classes

La classe `java.lang.ClassLoader` est une classe qui charge toutes les classes dans la machine virtuelle Java. Son implémentation de base (ce qui sous-entend qu'elle peut avoir plusieurs implémentations) peut charger des classes Java à partir de fichiers `.class`, `.zip` ou `.jar`, qu'elles soient locales ou distantes (*via* une URL sur le réseau). On peut donc charger dynamiquement une classe *via* un objet instance de la classe `ClassLoader` :



#### Exemple

```
ClassLoader loader = this.getClass().getClassLoader();  
MyType main = (MyType)loader.loadClass("MaClasse",  
    true).newInstance();
```

### 3.4.2 Seconde solution de chargement dynamique de classes

*Via* la méthode statique `forName()` de la classe `Class` (comme cela a été fait avec JDBC) : `static Class.forName(String className)`

- Son argument est le nom de la classe et non pas le nom du fichier dans lequel se trouve la classe (il faut donc penser à bien positionner la variable `ClassPath` pour que la JVM trouve le fichier contenant la classe à charger),
- Cette méthode retourne une référence vers une classe (vue comme un objet instance de la classe `Class`) qu'il est ensuite possible d'instancier *via* la méthode `newInstance()` de la classe `Class` (qui retourne un `Object`) ; ceci n'est possible que si cette classe possède un constructeur sans argument.



#### Exemple

```
Class.forName("com.mysql.jdbc.Driver").newInstance();
```

- La classe est chargée dans l'espace de nommage courant au `ClassLoader` qui crée un nouvel espace de nommage associé à cet objet (`ClassLoader`).

## 3.5 Exemples d'utilisation

### 3.5.1 Exemple n°1

En Java, l'archive `tools.jar` (en standard dans le J2SE) contient les outils liés au compilateur. Ce sont des wrappers qui encapsulent les outils `javac`, `jdb`, `java`, `rmic`, etc.



#### Exemple

```
// Un programme qui affiche toutes les données members
// d'un objet. On suppose que c est un objet de
// type Class.
Field[] publicFields = c.getFields();
System.out.println("Données membres : ");

for (int i = 0; i < publicFields.length; i++)
{
    String fieldName = publicFields[i].getName();
    Class typeClass = publicFields[i].getType();
    String fieldType = typeClass.getName();
    System.out.println(fieldName + "/" + fieldType);
}
```

### 3.5.2 Exemple n°2



#### Exemple (début)

```
import java.io.*;
import java.util.*;
import java.lang.reflect.*;

// API contenue dans la bibliothèque tools.jar.
import com.sun.tools.javac.Main;

public class Compile
{
    String ClassName = "MaClass";
    String ClassSource = "MaClass.java";

    public boolean compilation()
    {
        String[] source = new String(ClassSource);
        int result =
            com.sun.tools.javac.Main.compile(source);
        return (result == 0);
    }
}
```



### Exemple (suite)

```
public void creation ()
{
    try
    {
        // Crée le fichier dans un répertoire visible
        // du ClassLoader courant.
        FileWriter aWriter =
            new FileWriter(ClassSource, true);

        // Génération du contenu
        aWriter.write("/* ... blablabla ... */\n");
        aWriter.write("public class MaClass {");
        aWriter.write("public void hello () {");
        aWriter.write("System.out.println(\"Hello! \");");
        aWriter.write("}}\n");
        aWriter.flush();
        aWriter.close();
    }
    catch (Exception e)
    { e.printStackTrace(); }
}

public void lancement()
{
    try
    {
        // Types des paramètres de la méthode à invoquer.
        Class[] params = {};
        // Valeur de ces paramètres.
        Object[] paramsObj = {};

        // Utilise l'API Reflection pour charger
        // la classe. Charge le nouveau type
        // (i.e. la nouvelle classe).
        Class thisClass = Class.forName(ClassName);
        Object iClass = thisClass.newInstance();

        // Invoque la méthode 'hello' de la classe
        // dynamiquement créée.
        Method thisMethod = thisClass.getDeclaredMethod
            ("hello", params);
        thisMethod.invoke(iClass, paramsObj);
    }
    catch (Exception e)
    { e.printStackTrace(); }
}
```





### Exemple (fin)

```
public static void main (String[] args)
{
    Compile mtc = new Compile();

    // On génère à la volée le code source Java
    //du programme.
    mtc.creation();

    // Puis on le compile.
    if (mtc.compilation())
    {
        System.out.println("Running " +
            mtc.ClassName + " :\n\n");

        // Et enfin on l'exécute.
        mtc.lancement();
    }
    else
        System.out.println(mtc.ClassSource +
            " contient des erreurs de compilation.");
}
```

## 4 Processus concurrents

Les **threads** permettent le déroulement de plusieurs traitements de façon simultanée. On les appelle « processus légers » car, à la différence des processus, ils partagent le même adressage mémoire. Par conséquent, un objet créé par un *thread* pourra être accessible par un autre *thread* (du même processus).

Au niveau matériel, un seul *thread* peut occuper le processeur (sauf sur les processeurs *hyperthreading*), l'exécution des autres *threads* est alors suspendue. C'est ce que l'on appelle « l'accès concurrent » (les *threads* se partagent l'accès au processeur). Une fois le temps processeur alloué au *thread* actif écoulé, celui-ci est alors suspendu et un autre *thread* est activé ou réactivé par le processeur. Un *thread* actif peut être suspendu même s'il n'a pas fini son exécution.

Le but est :

- De donner l'impression, au niveau de l'application, que les différents *threads* qui la composent s'exécutent en parallèle,
- Alors qu'au niveau du processeur les *threads* sont en réalité exécutés en alternance.

L'objectif est ici de permettre à un programme de réaliser plusieurs opérations « simultanément ». Ceci offre de nombreux avantages, notamment pour créer des applications robustes :

- On peut créer des processus « séparés » qui exécuteront chacun une tâche précise sans bloquer le déroulement du programme.
- Les *threads* permettent d'exploiter au mieux les ordinateurs dotés de plusieurs processeurs (logiques ou réels).
- L'utilisation des *threads* masque la complexité des mécanismes de *time slicing* car tout est géré par la JVM.
- Les *threads* permettent d'accéder à la programmation concurrente (parallèle) pour que plusieurs *threads* puissent travailler sur les mêmes données. Il faudra dans ce cas implémenter des mécanismes de synchronisation.
- Les *threads* permettent de gérer des interfaces utilisateurs sophistiquées et la gestion de graphiques animés.
- On pourra éviter le blocage du programme par certains appels bloquants : création de serveurs *multithreads* capables de traiter plusieurs connexions simultanées, ...

### 4.1 Les *threads* en Java

Quand on lance un programme Java *via* la JVM, on lance en fait un processus contenant plusieurs *threads* :

- Le *thread* principal (celui qui exécute le code à partir du `main()`),
- Un *thread* pour la gestion du *garbage collector*,
- ...

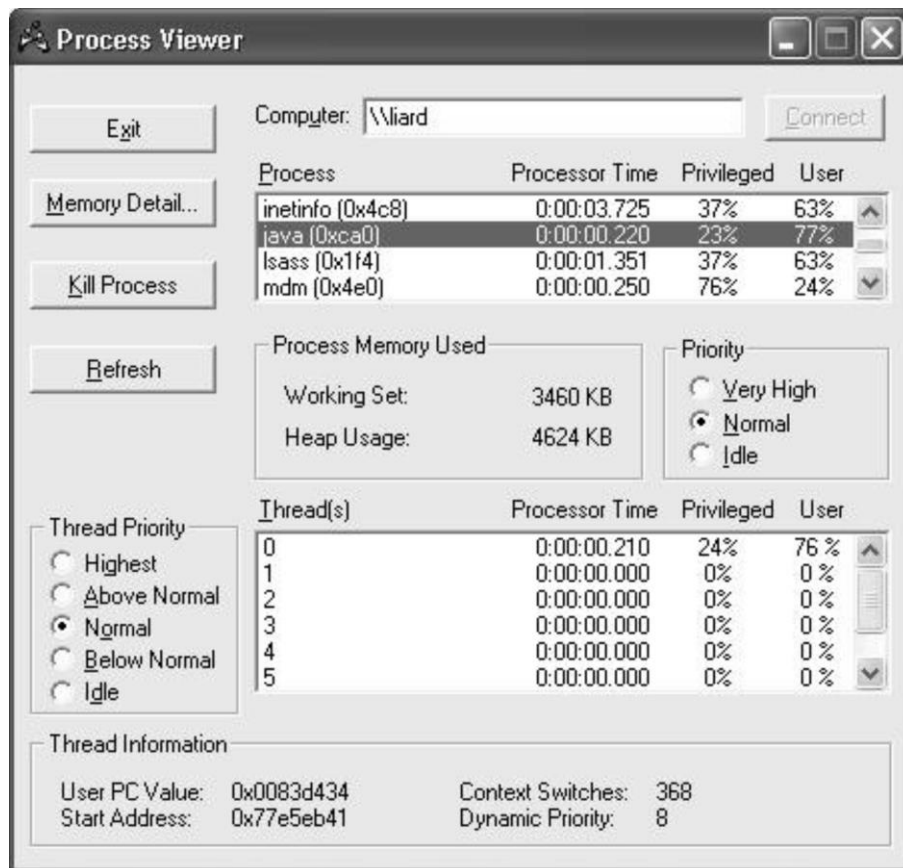


Figure 8 : threads d'un processus Java faisant une boucle infinie (`while(true)`)

Il est bien sûr possible de créer et de manipuler explicitement des *threads* en Java. Le langage Java offre 2 mécanismes permettant de faire de la programmation concurrente :

- La classe **Thread**,
- L'interface **Runnable**.

Chacune de ces solutions a ses avantages et ses inconvénients.

## 4.2 La classe Thread

En Java, toute classe héritant de la classe **Thread** va :

- Pouvoir être « threadée »,
- Hériter implicitement de méthodes permettant de contrôler le « *threading* ».

Ainsi, les instances de classes héritant de la classe **Thread** peuvent constituer autant de *threads*.

#### 4.2.1 Activer un *thread*

Les méthodes `start()` et `run()` héritées de la classe `Thread` sont chargées de s'assurer du « lancement » du code threadé.

La méthode `run()` est toujours public et doit être redéfinie si vous désirez que votre *thread* fasse quelque chose. C'est cette méthode qui va contenir tout le bloc d'instructions qui sera exécuté par le *thread*.

La méthode `run()` sera lancée par la méthode `start()` (et uniquement par celle-ci : c'est elle qui crée le *thread* et lui fait exécuter le code contenu dans la méthode `run()`).

##### Exemple



```
public class MonThread extends Thread
{
    public void run()
    {
        // Code éventuel du thread.
    }

    public static void main(String[] args)
    {
        MonThread t = new MonThread();

        t.start();
        // Exécute la méthode run() du thread 't'.
    }
}
```

#### 4.2.2 Arrêter un *thread*

Pour provoquer l'arrêt d'un *thread*, il est conseillé de faire une vérification périodique d'une de ses variables, généralement de type `boolean`, que l'on mettra à `true` pour signaler qu'il est nécessaire de stopper le *thread*. Ainsi, en testant et prévoyant convenablement les conditions d'arrêt du *thread*, on pourra libérer proprement les ressources qu'il occupe ainsi que les verrous qu'ils auraient posés (voir plus loin) et terminer l'exécution de la méthode `run()`.

Si le *thread* à arrêter procède à des attentes (voir plus loin), il ne sera pas toujours possible de procéder de cette manière. Dans ce cas-là, on peut avoir recours à la méthode `interrupt()` qui provoque la levée d'une exception `InterruptedException`. La gestion de cette exception permet de libérer proprement les ressources du *thread* dans un bloc `try/catch` avant de terminer la méthode `run()`.

De même, si le *thread* à arrêter est bloqué sur une opération d'entrée/sortie sur un canal interruptible (`java.nio.channels.InterruptibleChannel`) il est aussi possible d'utiliser la méthode `interrupt()` qui fermera celui-ci et lèvera une exception `ClosedByInterruptException`. Le traitement de cette dernière dans un bloc `try/catch` permettra de libérer proprement les ressources et de terminer la méthode `run()`.

Il existe une méthode **stop()** qui n'a été conservée que pour des raisons de compatibilité. Ne l'utilisez jamais : elle provoque l'arrêt brutal du thread et ne permet pas de vérifier que toutes les opérations de libération de ressources et de verrous s'effectuent correctement.

### 4.2.3 Suspendre un *thread*

La méthode statique **Thread.sleep(long delai)** permet de suspendre l'exécution du thread courant pendant **delai** ms, à la suite de quoi l'exécution du *thread* reprendra. Une exception **InterruptedException** sera levée si un appel de la méthode **interrupt()** est fait alors que le *thread* est suspendu avec **sleep()**.

#### Exemple (début)

```
public class MonThread extends Thread
{
    public void run()
    {
        try
        {
            System.out.println("Début du thread,
                               attente de 7 secondes...");
            Thread.sleep(7000);
            System.out.println("7 sec. se sont écoulées.");
        }
        catch (InterruptedException ie)
        {
            System.out.println("Réveillé avant la fin
                               du délai !");
        }
        finally
        { System.out.println("Fin du thread."); }
    }

    public static void main(String[] args)
    {
        try
        {
            MonThread t = new MonThread();

            t.start();
            Thread.sleep(3000);
            System.out.println("On interrompt le thread
                               après seulement 3 secondes.");
            t.interrupt();
        }
        catch (InterruptedException ie)
        { ie.printStackTrace(System.err); }
    }
}
```





### Exemple (fin)

L'affichage provoqué par ce code est le suivant :

```
Début du thread, attente de 7 secondes...
<Pause de 3 secondes>
On interrompt le thread après seulement 3 secondes.
Réveillé avant la fin du délai !
Fin du thread.
```

On peut remarquer que la méthode statique **Thread.sleep()** peut être appelée à d'autres endroits que dans le code de la méthode **run()** d'un *thread*, comme ici dans la méthode **main()** de l'exemple. Le « programme principal » peut être vu comme le *thread* principal du processus.

#### 4.2.4 Attendre la fin d'un *thread*

La méthode **join()** fait attendre la fin d'un *thread* au *thread* en cours d'exécution. Elle permet donc de s'assurer que le traitement d'un *thread* est terminé avant de poursuivre le déroulement du *thread* courant.

### Exemple (début)



```
public class MonThread extends Thread
{
    public void run()
    {
        try
        {
            System.out.println("Début du long traitement.");
            Thread.sleep(3000);
            System.out.println("Fin du long traitement.");
        }
        catch (InterruptedException ie)
        { ie.printStackTrace(System.err); }
    }

    public static void main(String[] args)
    {
        try
        {
            MonThread t = new MonThread();

            t.start();
            System.out.println("Attente de la fin de t.");
            t.join();
            System.out.println("Le thread t a fini.");
        }
        catch (InterruptedException ie)
        { ie.printStackTrace(System.err); }
    }
}
```

**Exemple (fin)**

L'affichage provoqué par ce code est le suivant :

```
Attente de la fin de t.  
Début du long traitement.  
<Pause de 3 secondes>  
Fin du long traitement.  
Le thread t a fini.
```

Tout comme la méthode **Thread.sleep()**, la méthode **join()** peut lever une **InterruptedException**. Ceci arrive quand :

- Un *thread* **t1** n'a pas fini son exécution,
- Un *thread* **t2** appelle **t1.join()**,
- **t2.interrupt()** est appelée.

Il est également possible de spécifier le temps maximal d'attente à l'aide des surcharges suivantes :

- **join(long delai)** : attend au maximum **delai** ms,
- **join(long milli, int nano)** : attend au maximum **milli** ms + **nano** ns (risque d'**IllegalArgumentException** si la valeur de **nano** est incorrecte).

#### 4.2.5 Tester l'exécution d'un *thread*

La méthode **isAlive()** permet de savoir si un *thread* est toujours « en vie » (renvoie **true**) ou non (renvoie **false**). On considère qu'un *thread* est dans l'état *alive* s'il a démarré (sa méthode **start()** a été appelée) et qu'il n'est pas encore terminé (*i.e.* pas encore dans l'état *dead*, sa méthode **run()** n'étant pas terminée).

#### 4.2.6 Priorité d'exécution d'un *thread*

Il est possible de changer la priorité d'exécution des *threads* afin de faire en sorte que certains d'entre eux s'exécutent « plus vite » que d'autres. La priorité d'un *thread* est définie par un entier variant entre 1 (priorité la plus faible) et 10 (priorité la plus haute). La classe **Thread** possède 3 variables statiques de priorité :

- **MIN\_PRIORITY** correspond à la priorité 1,
- **NORM\_PRIORITY** correspond à la priorité 5
- **MAX\_PRIORITY** correspond à la priorité 10.

On peut changer la priorité d'un *thread* grâce à la méthode **setPriority(int priorite)**. On peut récupérer la priorité courante d'un *thread* grâce à la méthode **int getPriority()**.



## Exemple

Impact de la priorité sur un thread

```
public class ThreadPriority extends Thread
{
    private long counter = 0;
    private static boolean arreterThreads = false;

    public void run()
    { while(ThreadPriority.arreterThreads == false)
      counter++; }

    public long getCounter()
    { return (this.counter); }

    public static void main(String[] args)
    throws Exception
    {
        ThreadPriority thread1 = new ThreadPriority();
        ThreadPriority thread2 = new ThreadPriority();
        ThreadPriority thread3 = new ThreadPriority();

        thread1.setPriority(Thread.MIN_PRIORITY);
        thread2.setPriority(Thread.NORM_PRIORITY);
        thread3.setPriority(Thread.MAX_PRIORITY);
        thread1.start();
        thread2.start();
        thread3.start();
        Thread.sleep(5000);
        ThreadPriority.arreterThreads=true;
        thread1.join();
        thread2.join();
        thread3.join();
        System.out.println("Thread 1 : counter == "
            + thread1.getCounter());
        System.out.println("Thread 2 : counter == "
            + thread2.getCounter());
        System.out.println("Thread 3 : counter == "
            + thread3.getCounter());
    }
}
```

Résultats :

	Thread 1	Thread 2	Thread 3
	(priorité normale)	(priorité normale)	(priorité normale)
Test 1	250 752 409	256 697 243	251 964 807
	(priorité minimale)	(priorité normale)	(priorité maximale)
Test 2	11 104 663	20 673 290	1 164 460 398

### 4.3 L'interface `Runnable`

L'autre façon de « threader » une classe est de lui faire implémenter l'interface **`Runnable`**. Ceci oblige à redéfinir la méthode **`run()`** dans la classe implémentant cette interface.


L'intérêt d'utiliser l'interface **`Runnable`** plutôt qu'une extension de la classe **`Thread`** est que la première solution permet à une classe « threadée » d'hériter d'une classe autre que **`Thread`** puisque Java interdit l'héritage multiple. Cela permet aussi de partager des données entre *threads* sans que celles-ci soient **`static`**.

#### 4.3.1 Créer un *thread* via un objet `Runnable`

L'exécution d'un *thread* qui implémente l'interface **`Runnable`** diffère quelque peu de celle d'un *thread* qui hérite de la classe **`Thread`**. Ici, il faut créer une instance de la classe **`Thread`** avec comme paramètre l'instance de la classe qui implémente **`Runnable`**. À noter qu'on peut ainsi créer plusieurs *threads* à partir d'une même instance d'une classe implémentant **`Runnable`**.

Enfin, il faut noter qu'il est tout à fait possible de créer une classe implémentant **`Runnable`** qui se lance automatiquement au moment de son instantiation mais cela fait perdre l'intérêt de cette interface au niveau des partages d'informations entre *threads* (voir plus loin).

#### Exemple (début)



```
// Horloge.java
import java.util.*;
import java.text.*;
import java.awt.*;

public class Horloge extends Label implements Runnable
{
    private DateFormat timeFormat
        = DateFormat.getTimeInstance();

    public void run()
    {
        try
        {
            while(true)
            {
                this.setText(timeFormat.format(new Date()));
                Thread.sleep(1000);
            }
        }
        catch(Exception exception)
        { exception.printStackTrace(); }
    }
}
```

### Exemple (fin)

```
public Horloge()
{
    this.setText(timeFormat.format(new Date()));
    this.setAlignment(Label.CENTER);

    // Création d'un thread basé sur l'horloge créée.
    (new Thread(this)).start();
}
}
```

```
// Start.java
import java.awt.*;
import java.applet.*;

public class Start extends Applet
{
    public void init()
    {
        this.setLayout(new FlowLayout());

        // Première horloge (avec son propre thread).
        Label clock1 = new Horloge();
        clock1.setBackground(new Color(200,200,255));

        // Deuxième horloge (avec son propre thread).
        Label clock2 = new Horloge();
        clock2.setBackground(new Color(200,255,200));

        // Troisième horloge (avec son propre thread).
        Label clock3 = new Horloge();
        clock3.setBackground(new Color(255,200,200));

        this.add(clock1);
        this.add(clock2);
        this.add(clock3);
    }
}
```



### Exemple

Création de plusieurs *threads* depuis un seul objet :

```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 5; i++)
            System.out.println(i);
    }

    public static void main(String[] args)
    {
        Thread t1, t2;

        MonImplDeRunnable ex = new MonImplDeRunnable();
        t1 = new Thread(ex);
        t2 = new Thread(ex);
        t1.start();
        t2.start();
    }
}
```



Affichage provoqué :

```
0
1
2
0
1
3
4
2
3
4
```


Ce résultat peut varier car la distribution du temps processeur entre les *threads* et le processus principal (celui qui exécute la méthode **main()**) dépend de beaucoup de facteurs. De plus, les 2 *threads* créés ne démarrent pas en même temps mais l'un après l'autre.

### 4.3.2 Accès aux *threads* créés à partir d'objets `Runnable`

Lorsqu'une classe implémente **Runnable**, on n'a pas directement accès aux méthodes de la classe **Thread** pour contrôler le *thread* en cours d'exécution. En effet, l'objet passé en paramètre du constructeur **Thread(Runnable r)** n'est pas le *thread* qui sera exécuté mais servira juste de base à celui-ci.

Pour y avoir accès, on peut utiliser la méthode statique **Thread.currentThread()** qui retourne le *thread* en cours d'exécution. On pourra alors utiliser les méthodes de la classe **Thread** sur le *thread* renvoyé par l'appel de cette méthode statique.

#### Exemple




```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        System.out.println("Mon nom est "
            + Thread.currentThread().getName());
    }

    public static void main(String[] args)
    {
        new Thread(new MonImplDeRunnable()).start();
    }
}
```

#### Exemple

Équivalent avec la classe **Thread** :



```
public class MonThread extends Thread
{
    public void run()
    {
        System.out.println("Mon nom est " + this.getName());
    }

    public static void main(String[] args)
    {
        new MonThread.start();
    }
}
```

#### 4.4 Choisir la classe `Thread` ou l'interface `Runnable` ?

Le tableau suivant récapitule les avantages et inconvénients de ces 2 possibilités :

	Avantages	Inconvénients
<b>extends</b> <code>java.lang.Thread</code>	Chaque <i>thread</i> a ses données qui lui sont propres.	On ne peut plus hériter d'une autre classe.
<b>implements</b> <code>java.lang.Runnable</code>	L'héritage reste possible. En effet, on peut implémenter autant d'interfaces que l'on souhaite.	Les attributs de votre classe sont partagés pour tous les <i>threads</i> qui y sont basés. Dans certains cas, il peut s'avérer que cela soit un atout.

Tableau 2 : choisir la classe `Thread` ou l'interface `Runnable` ?

#### 4.5 Cycle de vie d'un *thread*

Un *thread* est caractérisé par son état :

- Avant que la méthode `start()` ne soit appelée, le *thread* est dans l'état *new*.
- À l'appel de la méthode `start()`, le *thread* rentre dans l'état *runnable*.
- À partir de cet instant, le *thread* est considéré comme *alive*, et ce jusqu'à ce qu'il soit *dead*.
- Quand la JVM accorde du temps processeur à un *thread*, il est considéré dans l'état *running*.
- Lorsque le *thread* a fini d'exécuter la méthode `run()`, il entre dans l'état *dead* et ne peut pas être relancé par la méthode `start()` (cependant, l'instance est toujours présente, ce qui permet d'avoir des informations sur le *thread*).

Un *thread* peut ainsi passer par différents états, tout au long de son cycle de vie :

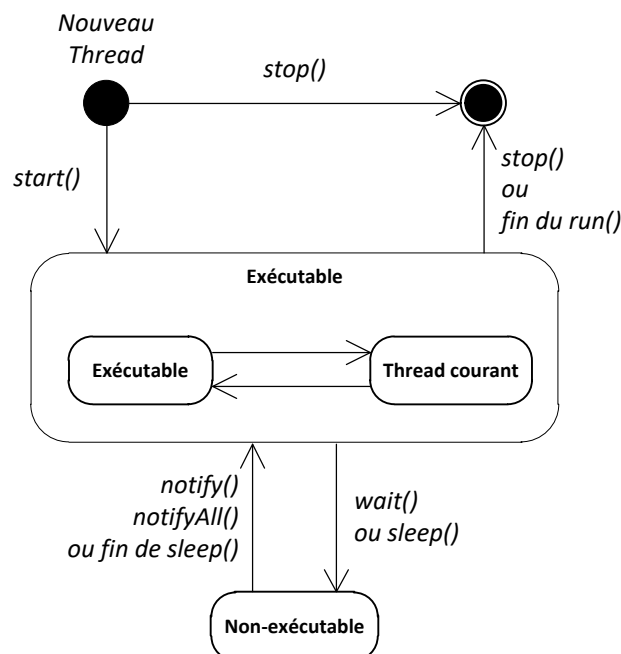


Figure 9 : cycle de vie d'un *thread*

Quand on crée un *thread*, celui-ci est par défaut dans son **état initial**. Il faut invoquer la méthode **start()** pour lui permettre de passer dans un **état exécutable**. Si on invoque la méthode **stop()**, le *thread* passe dans un **état terminal** sans qu'il n'ait eu le temps d'exécuter la moindre instruction. Pour terminer un *thread*, deux techniques sont proposées : soit on sort normalement de sa méthode **run()**, soit on invoque sur le *thread* la méthode **stop()**. Dans les deux cas, le *thread* ne pourra jamais redémarrer.

Tant que le *thread* est dans un état exécutable, il est susceptible de pouvoir être traité par le processeur. Celui-ci lui sera octroyé durant un bref instant (quelques millisecondes). Au terme de cette durée, le système élira un autre *thread* exécutable et lui donnera la main.

Les systèmes d'exploitations modernes sont qualifiés de préemptifs : aucune action n'est à faire par le *thread* pour que le système lui ôte le processeur. Mais il existe aussi d'autres systèmes qualifiés de non préemptifs : dans ce cas les choses se compliquent un peu. Il est de la charge du *thread* d'exécuter un appel à la méthode **yield()** pour pouvoir donner la main à un autre *thread*. Nous ne traiterons pas ici de cette technique.

Un *thread* peut aussi être sorti de la liste des *threads* en attente du processeur. Dans ce cas, il passe dans un état **non exécutable**. Il lui faudra rejoindre le groupe des *threads* exécutables pour pouvoir à nouveau recevoir le processeur. Pour passer dans l'état non exécutable, deux méthodes peuvent être invoquées : **Thread.sleep(long duree)**, qui suspend le *thread* durant quelques instants, ou les méthodes **wait()** de la classe **java.lang.Object** (nous reviendrons ultérieurement sur ces dernières).

## 4.6 Groupe de *threads*

Une possibilité intéressante consiste à regrouper différents *threads* au sein d'un groupe de *threads*. Pour inscrire un *thread* dans un groupe, il faut déjà avoir créé le groupe. Pour ce faire, il faut instancier un objet de la classe **ThreadGroup**. Une fois le groupe créé, on peut attacher des *threads* ou d'autres groupes de *threads* à ce groupe. L'attachement d'un *thread* à un groupe se fait *via* certains constructeurs de la classe **Thread**.

Par la suite, un *thread* ne pourra en aucun cas changer de groupe. Si vous avez mis en œuvre de l'héritage, n'oubliez pas que dans ce cas tous les constructeurs parents sont invoqués lors de la création d'un objet.

Une fois des *threads* attachés à un groupe, on peut alors invoquer les méthodes de contrôle d'exécution des *threads* sur les objets du groupe. Les noms des méthodes sont identiques à celles de la classe **Thread** : **suspend()**, **resume()**, **stop()**, ...

## 4.7 Partage d'informations entre *threads*


Il existe 2 manières de partager des données entre *threads* :

- En utilisant des variables **static** (c'est la solution la plus simple),
- En créant des objets **Thread** à partir d'un seul et même objet **Runnable** afin que ces objets **Thread** aient des informations en commun (celles de l'objet **Runnable** à partir duquel ils ont été créés).

### 4.7.1 Partage d'informations *via* des variables statiques

Une variable statique étant une variable de classe, elle est partagée par toutes les instances de cette classe. Donc, définir des variables statiques dans une classe héritant de la classe **Thread** permettra de partager les valeurs de ces variables entre tous les *threads* créés à partir de cette classe de façon très simple.

#### Exemple



```
public class MonThread extends Thread
{
    static int a;

    public void run()
    {
        for (int i = 0; i < 5; i++)
            System.out.println(this.getName() + " : " + a++);
    }

    public static void main(String[] args)
    {
        new MonThread().start();
        new MonTrhead().start();
    }
}
```

Affichage provoqué :

```
Thread-0 : 0
Thread-0 : 1
Thread-0 : 2
Thread-0 : 3
Thread-0 : 4
Thread-1 : 5
Thread-1 : 6
Thread-1 : 7
Thread-1 : 8
Thread-1 : 9
```

Il s'agit d'une possibilité d'affichage, la distribution du temps processeur pouvant varier.




#### 4.7.2 Partage d'informations *via* un objet `Runnable`

Utiliser des variables statiques apporte une grande facilité d'implémentation des accès partagés. Cependant, cette solution peut ne pas correspondre à vos besoins dans certains cas où vos classes ne sont pas destinées qu'à être utilisées en *threads* concurrents.

Ainsi, il peut être nécessaire de ne faire partager des données qu'entre *threads* 2-à-2 (ou plus) sans pour autant que tous les *threads* ne partagent les mêmes variables. Le constructeur `Thread(Runnable r)` permet d'implémenter cette solution puisqu'il est possible de créer plusieurs *threads* à partir d'un même objet **Runnable**, implicitement ces *threads* partagent les données de l'objet **Runnable** à partir duquel ils ont été créés.

##### Exemple (début)



```
public class MonImplDeRunnable implements Runnable
{
    private int a, b;
    private String text;

    MonImplDeRunnable(String text)
    { this.text = text; }

    public void run()
    {
        for (int i = 0; i < 3; i++)
        {
            System.out.println(this.text + " "
                               + Thread.currentThread().getName()
                               + " a : " + a++);
            System.out.println(this.text + " "
                               + Thread.currentThread().getName()
                               + " b : " + b++);
        }
    }

    public static void main(String[] args)
    {
        MonImplDeRunnable o1, o2;

        o1 = new MonImplDeRunnable("Groupe 1");
        new Thread(o1).start();
        new Thread(o1).start();

        o2 = new MonImplDeRunnable("Groupe 2");
        new Thread(o2).start();
        new Thread(o2).start();
    }
}
```

**Exemple (fin)**

Affichage provoqué :



```
Groupe 1 Thread-0 a : 0
Groupe 1 Thread-0 b : 0
Groupe 1 Thread-0 a : 1
Groupe 1 Thread-1 a : 2
Groupe 1 Thread-1 b : 1
Groupe 2 Thread-3 a : 1
Groupe 2 Thread-3 b : 0
Groupe 2 Thread-2 a : 0
Groupe 2 Thread-2 b : 1
Groupe 1 Thread-0 b : 2
Groupe 1 Thread-0 a : 4
Groupe 2 Thread-3 a : 2
Groupe 2 Thread-3 b : 2
Groupe 1 Thread-1 a : 3
Groupe 1 Thread-1 b : 4
Groupe 1 Thread-0 b : 3
Groupe 2 Thread-2 a : 3
Groupe 1 Thread-1 a : 5
Groupe 1 Thread-1 b : 5
Groupe 2 Thread-3 a : 4
Groupe 2 Thread-3 b : 4
Groupe 2 Thread-2 b : 3
Groupe 2 Thread-2 a : 5
Groupe 2 Thread-2 b : 5
```

Il s'agit d'une possibilité d'affichage, la distribution du temps processeur pouvant varier.

Étant donné que la classe **Thread** implémente l'interface **Runnable**, il est donc possible de passer une instance de la classe **Thread** en paramètre du constructeur de la classe **Thread**. Comme dans le cas général de *threads* créés à partir d'un objet **Runnable**, même dans ce cas particulier on doit utiliser la méthode statique **Thread.currentThread()** pour accéder au *thread* en cours d'exécution.

#### 4.7.3 Comparaison des techniques de partage d'informations

L'utilisation des **membres statiques** permet de spécifier quelles variables membres on désire partager. Les autres données membres ne seront pas partagées. L'inconvénient est que tous les objets issus de la classe partageront toutes ses données statiques, même des objets nouvellement créés, ce qui fait perdre un avantage de la POO.

L'utilisation du constructeur **Thread(Runnable r)** offre l'avantage de ne pas partager les informations entre toutes les instances de la classe mais seulement entre les *threads* issus d'un même objet **Runnable**. Les inconvénients sont que toutes les variables membres sont partagées et qu'il est nécessaire de créer deux objets au lieu d'un seul pour créer ne serait-ce qu'un seul *thread*.

## 4.8 Synchronisation/verrouillage

Quand plusieurs *threads* travaillent en parallèle, il se peut qu'ils aient accès aux mêmes données et qu'ils les modifient de manière non cohérente. Pour résoudre ce problème, on peut bloquer l'accès à certaines données par un système de verrouillage afin de pouvoir respecter une certaine cohérence dans le bon déroulement du programme.

Par exemple, deux *threads* **t1** et **t2** cherchent à incrémenter la valeur d'un attribut statique de type entier nommé **MaClasse.shared** ayant initialement la valeur 0. Seul un *thread* à la fois peut exécuter du code, mais n'oublions pas que les systèmes les plus modernes sont préemptifs ! De plus, une instruction telle que **MaClasse.shared = MaClasse.shared + 1** ; se traduit en plusieurs instructions en langage machine.

Imaginons que le premier *thread* **t1** évalue l'expression **MaClasse.shared + 1** (à la valeur 1 donc) mais que le système lui hôte le CPU juste avant l'affectation au profit du second *thread* **t2**. Ainsi, la variable **MaClasse.shared** vaut toujours 0.

Le *thread* **t2** se doit lui aussi d'évaluer la même expression. Il évalue **MaClasse.shared + 1** (à la valeur 1 également puisque la variable **MaClasse.shared** vaut toujours 0) mais, là encore, le système redonne la main au premier *thread* avant que l'affectation ne soit effectuée. La variable **MaClasse.shared** vaut donc toujours 0.

Le *thread* **t1** finalise l'instruction en effectuant l'affectation : la variable **MaClasse.shared** vaut donc maintenant 1. Le *thread* **t2** fait de même : la variable **MaClasse.shared** vaut alors toujours 1 puisque, là encore, le *thread* affecte à cette variable la valeur 1.

Au final de ce scénario, l'entier n'aura été incrémenté que d'une seule et unique unité alors qu'il aurait dû être incrémenté de 2 unités. De tels scénarios peuvent amener à des comportements d'applications chaotiques. Il est donc vital d'avoir à disposition des mécanismes de synchronisation/verrouillage.

### 4.8.1 Synchronisation d'un bloc d'instructions

Ceci consiste à poser un verrou sur un objet pendant l'exécution d'un certain nombre d'instructions. La portée de ce verrou est définie au sein du *thread* par un bloc identifié par le mot-clé **synchronized** associé à une référence vers l'objet à verrouiller. Les autres *threads* ne pourront pas effectuer d'action sur cet objet et attendront la levée du verrou avant de poursuivre leurs traitements.

Pour que la pose du verrou soit effective entre différents *threads*, il faut bien sûr que l'objet verrouillé soit partagé entre ces *threads*. Si ce n'est pas le cas, il y aura pose de verrou sur autant d'objets différents, rendant l'utilisation du bloc **synchronized** inutile.

### Exemple

L'exemple suivant montre l'utilisation de la synchronisation d'un bloc d'instructions. Dans la méthode `run()`, il y a une synchronisation sur la variable `i` de type `Integer`. Pour bien mettre en évidence la synchronisation, on utilise la méthode `sleep()` qui va suspendre l'exécution du *thread* pendant 100 ms. Par conséquent, l'objet `i` sera inaccessible pendant une durée plus longue afin de pouvoir constater l'effet produit.

```
public class Synchronisation extends Thread
{
    static Integer i;
    int max;
    String nom;

    public Synchronisation(String nom, int max)
    {
        i = new Integer(0);
        this.max = max;
        this.nom = nom;
    }

    public void run()
    {
        synchronized(i)
        {
            while (i.intValue() < max)
            {
                try
                { sleep(100); }
                catch (Exception e)
                { System.err.println(e); }

                System.out.println(nom + " modifie i.");
                i = new Integer(i.intValue() + 1);
                System.out.println(i.intValue());
            }
            System.out.println(nom + " est fini !");
        }
    }

    public static void main(String[] args)
    {
        new Synchronisation("t1", 50).start();
        new Synchronisation("t2", 100).start();
    }
}
```

À l'exécution on voit que le *thread* `t2` attend que le *thread* `t1` ait fini de s'exécuter pour pouvoir modifier la variable `i` : `t1` fait d'abord passer `i` de 0 à 50 puis `t2` fait passer `i` de 50 à 100.

### Exemple (début)

Afin de mieux comprendre les choses, nous allons étudier un petit exemple : celui-ci va permettre à plusieurs *threads* de tracer des pixels en parallèle. Mais pour que le programme se déroule normalement, il faut que l'appel à la méthode de tracé soit synchronisé. Sinon, il y aura des possibilités pour que certains pixels soient sautés. L'interface graphique de l'applet présente deux boutons et une zone de dessin. Les deux boutons permettent de lancer le tracé dans la zone de dessin. L'un des boutons lance le tracé sans synchronisation alors que l'autre le fait en synchronisant les *threads*.



```
import java.applet.*;
import java.awt.*;
import java.awt.event.*;

public class Synchronized extends Applet
    implements ActionListener, Runnable
{
    private Panel pnlButtonBar = new Panel();
    private Button btnStartNormal
        = new Button("Non synchronisé");
    private Button btnStartSynchronized
        = new Button("Synchronisé");
    private Canvas cnvGraphic = new Canvas();
    private Label lblStatus
        = new Label("Choisissez un mode d'exécution");

    private boolean syncDrawMode = false;
    private int x, y;

    public void init()
    {
        this.pnlButtonBar.setLayout(new FlowLayout());
        this.pnlButtonBar.add(this.btnStartNormal);
        this.pnlButtonBar.add(this.btnStartSynchronized);
        this.btnStartNormal.addActionListener(this);
        this.btnStartSynchronized.addActionListener(this);

        this.setLayout(new BorderLayout());
        this.add(this.pnlButtonBar, BorderLayout.NORTH);

        this.cnvGraphic.setBackground(Color.white);
        this.add(this.cnvGraphic, BorderLayout.CENTER);

        this.add(this.lblStatus, BorderLayout.SOUTH);
    }
}
```

Exemple (fin)



```
public void plot()
{
    this.cnvGraphic.getGraphics().drawLine(this.x,
        this.y, this.x, this.y);

    if (++this.x >= 300)
    {
        this.x=0;
        this.y++;
    }
}

public void run()
{
    while(this.y < 100)
    {
        if (syncDrawMode == true)
        {
            synchronized(this)
            {
                this.plot();
            }
        }
        else
            this.plot();
    }
}

public void actionPerformed(ActionEvent event)
{
    this.syncDrawMode = (event.getSource()
        == this.btnStartSynchronized );
    this.cnvGraphic.repaint();
    this.x = this.y = 0;

    (new Thread(this)).start();
    (new Thread(this)).start();
    (new Thread(this)).start();
}
}
```


#### 4.8.2 Synchronisation de méthodes

Les méthodes peuvent elles-aussi bénéficier de la synchronisation afin d'éviter que plusieurs *threads* n'y accèdent en même temps. Lorsqu'une méthode est déclarée avec le modificateur **synchronized**, des *threads* issus d'un même objet ne peuvent accéder simultanément à celle-ci.

Ainsi, si un *thread* est en train d'exécuter une méthode **synchronized** et qu'un autre *thread* issu du même objet que le premier essaie à son tour d'accéder à cette méthode, il devra attendre que le premier *thread* termine l'exécution de la méthode pour pouvoir l'exécuter à son tour. En effet, dans ce cas un verrou est automatiquement posé sur l'ensemble de la méthode lorsqu'un premier *thread* y fait appel avant l'exécution de la première instruction. Ce verrou est libéré après l'exécution de la dernière instruction de la méthode synchronisée.

Dans le cas d'une méthode statique, tous les *threads* issus de la classe sont affectés et ne peuvent exécuter simultanément la méthode. Aucun autre *thread* ne pourra exécuter une méthode **synchronized** tant qu'un verrou est actif, sous réserve que les différents *threads* sont issus du même objet ou que la méthode soit statique.

##### Exemple (début)



```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        for (int i = 0; i < 5; i++)
            move();
    }


    synchronized private void move()
    {
        String nom = Thread.currentThread().getName();
        System.out.println(nom + " entre.");
        System.out.println(nom + " sort.");
    }

    public static void main(String[] args)
    {
        MonImplDeRunnable o = new MonImplDeRunnable();
        new Thread(o).start();
        new Thread(o).start();
    }
}
```

À l'exécution, on observe qu'il n'y a qu'un seul *thread* à la fois qui entre dans la méthode, l'autre *thread* attendant que le verrou sur la méthode soit libéré. Si on retirait le verrou sur la méthode `move()`, il est fortement probable que l'on verrait un des 2 *threads* entrer dans cette méthode avant que l'autre n'en soit sorti.

**Exemple (fin)**

Affichage provoqué :



```
Thread-0 entre.  
Thread-0 sort.  
Thread-0 entre.  
Thread-0 sort.  
Thread-0 entre.  
Thread-0 sort.  
Thread-1 entre.  
Thread-1 sort.  
Thread-0 entre.  
Thread-0 sort.  
Thread-1 entre.  
Thread-1 sort.  
Thread-1 entre.  
Thread-1 sort.  
Thread-0 entre.  
Thread-0 sort.  
Thread-1 entre.  
Thread-1 sort.  
Thread-1 entre.  
Thread-1 sort.
```

## 4.9 Mécanisme de notification entre *threads*

Il est possible de faire attendre un *thread* par un autre sans pour autant que celui-ci n'ait besoin de se terminer comme c'est le cas avec la méthode `join()`. On fait appel, pour cela, aux méthodes `wait()`, `notify()` et `notifyAll()`. Il est à noter que ces méthodes ne sont pas définies dans la classe `Thread` mais dans la classe `Object`.

**Attention**

Ces méthodes ne peuvent être appelées que sur un objet qui a été verrouillé par un bloc `synchronized`. Dans le cas contraire, une exception `IllegalMonitorStateException` sera levée.

À noter également que les méthodes `wait()`, `notify()` et `notifyAll()` ne sont pas mises en attente lorsqu'elles sont appelées par un *thread* sur un objet verrouillé par un autre *thread*.

### 4.9.1 Attendre une notification : la méthode `wait()`

Cette méthode provoque la suspension de l'exécution du *thread* dans lequel s'exécutait le code courant. Le *thread* courant est donc suspendu et attend que l'objet par rapport auquel on a réalisé la suspension émette une notification afin de reprendre l'exécution. Il est possible de spécifier une durée maximale de suspension via `wait(long duree)`. À la fin de la durée, le *thread* est réactivé, qu'une notification ait été reçue ou non par l'objet par rapport auquel on a suspendu le *thread*.

Suite à l'appel de la méthode `wait()`, un *thread* est dans un état *waiting* (état d'attente). Un appel aux méthodes `notify()` ou `notifyAll()` sur l'objet par rapport auquel on a mis le *thread* en attente permet à celui-ci de sortir de cet état d'attente.



La méthode `wait()` libère temporairement le verrou sur l'objet synchronisé. Une fois que le *thread* est réactivé (à l'aide de `notify()`, de `notifyAll()` ou à la fin de la durée maximale d'attente), le verrou est acquis de nouveau.

#### 4.9.2 Notifier un *thread* : la méthode `notify()`

Cette méthode appelée sur un objet `o` réactive un *thread* en attente suite à l'appel de la méthode `wait()` sur cet objet `o`.

S'il y a plusieurs *threads* à réactiver (*i.e.* en attente en même temps sur le même objet `o`), le choix du *thread* à réveiller est arbitraire et dépend de l'implémentation de la JVM.

#### 4.9.3 Notifier tous les *threads* : la méthode `notifyAll()`

Cette méthode appelée sur un objet `o` réactive tous les *threads* ayant effectué un appel de la méthode `wait()` sur ce même objet `o`.

#### 4.9.4 Exemples de notification

##### Exemple 1 (début)



```
public class MonImplDeRunnable implements Runnable
{
    public void run()
    {
        Thread t = Thread.currentThread();
        // Représente le thread en cours d'exécution au
        // sein d'une implémentation de Runnable.

        try
        {
            System.out.println("Début de "
                               + t.getName() + ", appel à wait().");

            synchronized(t)
            { t.wait(); }

            System.out.println(t.getName()
                               + " a fini d'attendre.");
        }
        catch (Exception e)
        { e.printStackTrace(System.err); }
        finally
        { System.out.println("Fin de " + t.getName()); }
    }
}
```

### Exemple 1 (fin)

```
public static void main(String[] args)
{
    try
    {
        Thread t = new Thread(new MonImplDeRunnable());

        t.start();
        System.out.println("On attend 3 secondes...");
        Thread.sleep(3000);
        System.out.println("On réveille " + t.getName());

        synchronized(t)
        { t.notify(); }
    }
    catch (Exception e)
    { e.printStackTrace(System.err); }
    finally
    { System.out.println("Fin du 'main'."); }
}
```



Affichage provoqué :

```
On attend 3 secondes...
Début de Thread-0, appel à wait().
<Pause de 3 secondes>
On réveille Thread-0
Thread-0 a fini d'attendre.
Fin de Thread-0
Fin du 'main'.
```

À noter que, selon les exécutions, la méthode `main()` aurait pu se terminer avant le thread "**Thread-0**".

### Exemple 2 (début)

```
public class MonImplDeRunnable implements Runnable
{
    private Thread t;

    MonImplDeRunnable()
    { super(); }

    MonImplDeRunnable(Thread t)
    {
        super();
        this.t = t;
    }

    public void run()
    {
        Thread currentThread = Thread.currentThread();
        String nom = currentThread.getName();

        try
        {
            if (t == null)
            {
                // t n'est pas affecté, il s'agit de t1
                System.out.println(nom + " attend 3 secondes
                    avant de réveiller les autres.");
                Thread.sleep(3000);
                System.out.println(nom + " réveille les autres
                    threads.");

                synchronized(currentThread)
                { currentThread.notifyAll(); }
            }
            else
            {
                synchronized(t)
                {
                    System.out.println(nom + " entre en
                        attente de " + t.getName());
                    t.wait();
                }
            }
        }
        catch (Exception e)
        { e.printStackTrace(System.err); }
        finally
        { System.out.println("Fin de " + nom); }
    }
}
```



### Exemple 2 (fin)

```
public static void main(String[] args)
{
    MonImplDeRunnable m1, m2;
    Thread t1, t2, t3;

    m1 = new MonImplDeRunnable();
    t1 = new Thread(m1);
    m2 = new MonImplDeRunnable(t1);
    t2 = new Thread(m2);
    t3 = new Thread(m2);

    t1.start();
    t2.start();
    t3.start();
}
```

Affichage provoqué :

```
Thread-0 attend 3 secondes avant de réveiller les
autres.
Thread-1 entre en attente de Thread-0
Thread-2 entre en attente de Thread-0
<Pause de 3 secondes>
Thread-0 réveille les autres threads.
Fin de Thread-2
Fin de Thread-0
Fin de Thread-1
```

#### 4.9.5 Interblocage

Mal utilisées, les techniques de synchronisation peuvent être à l'origine d'un dysfonctionnement de vos applications « threadées ». Si un *thread A* attend un *thread B* qui lui-même attend le *thread A*, ces deux *threads* seront interbloqués et ne pourront continuer leur exécution. On parle alors d'interblocage (*deadlock*).

L'exécution de *threads* dépendant de la distribution du temps processeur, détecter un interblocage est difficile. Le seul fait de tester une fois les applications « threadées » se révélera donc très souvent insuffisant. Il faut veiller à ce qu'un interblocage ne puisse pas se produire dès la conception de l'application.

### Exemple (début)

```
// Exemple produisant un interblocage
// pendant 3 secondes.
public class Interblocage extends Thread
{
    private Thread t;

    public void setThread(Thread t)
    { this.t = t; }

    public void run()
    {
        try
        {
            System.out.println(this.getName() +
                               " entre en attente de " + t.getName());

            synchronized(this.t)
            { t.wait(); }

            System.out.println(this.getName() +
                               " a fini d'attendre " + t.getName());
        }
        catch (Exception e)
        { e.printStackTrace(System.err); }
    }

    public static void main(String[] args)
    {
        Interblocage t1 = new Interblocage();
        Interblocage t2 = new Interblocage();
        t1.setThread(t2);
        t2.setThread(t1);
        t1.start();
        t2.start();

        try
        {
            Thread.sleep(3000);
            System.out.println(t1.getName() + " et "
                               + t2.getName() + " sont en situation
                               d'interblocage ; on réactive "
                               + t1.getName());

            synchronized(t1)
            { t1.notify(); }
        }
        catch (Exception e)
        { e.printStackTrace(System.err); }
    }
}
```





#### Exemple (fin)

Affichage provoqué :

```
Thread-0 entre en attente de Thread-1
Thread-1 entre en attente de Thread-0
<Pause de 3 secondes>
Thread-0 et Thread-1 sont en situation d'interblocage ;
on réactive Thread-0
Thread-1 a fini d'attendre Thread-0
Thread-0 a fini d'attendre Thread-1
```

Cet exemple met en évidence la libération temporaire du verrou sur `t1` lors de la mise en attente avec la méthode `wait()` par `t2`. En effet, cela permet à la méthode `main()` de pouvoir synchroniser sur le *thread* `t1` alors qu'il était déjà synchronisé par le *thread* `t2`.

#### 4.9.6 Exemple de producteurs/consommateurs

Pour mettre en œuvre un exemple de synchronisation un peu évolué, nous allons considérer un cas d'école : un exemple de producteurs/consommateurs. Dans un tel cas, une ressource partagée est utilisée par des *threads* qui produisent et par des *threads* qui consomment. Mais attention, il est hors de question de consommer si rien n'a été produit.

Pour implémenter notre objet partagé, nous allons coder une pile qui sera bornée : pour empiler des données supplémentaires, il faudra attendre que des *threads* consommateurs aient dépilé les anciennes données. Dans ce cas, les choses sont subtiles. L'objet de synchronisation est clairement la pile. Nous allons donc, au gré de l'exécution du programme, endormir des *threads* sur cet objet. Mais deux types de *threads* seront à considérer : ceux qui produisent et ceux qui consomment.

Imaginons le scénario suivant : un *thread* empile une donnée. C'est donc un producteur et il va donc utiliser une méthode pour réveiller un éventuel consommateur. Mais qu'est ce qui garantit que le *thread* réveillé ne sera pas un autre producteur ? Si c'était le cas, nous aboutirions à des cas d'interblocage : le programme n'évoluerait plus, mais ne se terminerait pas.

Il nous faudra donc utiliser la méthode `notifyAll()` pour réveiller les *threads*. Il faut alors garantir que tous les *threads* réveillés qui n'ont pas accès à la ressource se rendorment rapidement. D'où le code des méthodes `push()` et `pop()` de la classe `Pile`.



#### Exemple (début)

```
public class Pile
{
    private int[] array;
    private int size, index;

    // Constructeurs de pile.
    public Pile()
    { this(5); }
}
```

### Exemple (fin)

```
public Pile(int size)
{
    this.size = size;
    this.index = 0;
    this.array = new int[size];
}

// Renvoie true si la pile est vide.
public boolean isEmpty()
{ return (index == 0); }

// Renvoie true si la pile est pleine.
public boolean isFull()
{ return (index == size); }


// Méthode synchronisée dépilant une valeur.
public synchronized int pop()
{
    try
    {
        while (this.isEmpty())
        {
            System.out.println("Consommateur Endormi");
            this.wait();
        }
    }
    catch(Exception e)
    { e.printStackTrace(); }
    int val = array[--index];
    this.notifyAll();
    return val;
}

// Méthode synchronisée empilant une valeur.
public synchronized void push(int value)
{
    try
    {
        while (this.isFull())
        {
            System.out.println("Producteur Endormi");
            this.wait();
        }
    }
    catch(Exception e)
    { e.printStackTrace(); }
    array[index++] = value;
    this.notifyAll();
}
}
```



Maintenant que la ressource partagée est prête, il ne nous reste plus qu'à coder les producteurs et les consommateurs. Dans les deux cas, ces deux types (classes) de composants partagent tous certaines caractéristiques : ils travaillent tous sur la même pile et dans les deux cas, cadencer les choses via un **Thread.sleep()** pourra permettre une bonne lisibilité des résultats sur la console. Nous utilisons ici l'héritage pour définir le tronc commun à tous nos *threads*. La classe de base se nomme **Fonctionneur** et elle est abstraite : nous ne voulons pas permettre d'instancier un quelconque objet de ce type là (de plus, nous n'avons pas d'implémentation de la méthode **run()** dans cette classe).

#### Exemple



```
abstract class Fonctionneur implements Runnable
{
    // La pile partagée par tous nos threads.
    private static Pile p = new Pile(5);

    // Un délai d'attente pour chaque thread.
    protected long sleepTime;

    // Un constructeur par défaut.
    // Pas de délai d'attente.
    public Fonctionneur ()
    { this(0); }

    // Un constructeur qui prend un délai
    // d'attente pour le thread.
    public Fonctionneur (long sleepTime)
    {
        this.sleepTime = sleepTime;
        (new Thread(this)).start();
    }

    // Permet de pouvoir récupérer la pile.
    public Pile getPile()
    { return p; }
}
```

Nous pouvons maintenant dériver de cette classe nos deux types de *threads* :

- Les producteurs (classe **Producteur**) qui vont produire des valeurs entières et donc les empiler (tant que cette dernière n'est pas pleine),
- Et les consommateurs (classe **Consommateur**) qui vont lire des valeurs entières sur la pile (tant que cette dernière n'est pas vide).



### Exemple

Les producteurs :



```
public class Producteur extends Fonctionneur
{
    private int value = 0;

    public Producteur()
    { super(); }

    public Producteur(long sleepTime)
    { super(sleepTime); }

    public void run()
    {
        while (true)
        {
            try
            {
                Thread.sleep((long)Math.random()
                    * this.sleepTime);
            }
            catch (InterruptedException e)
            { e.printStackTrace(); }

            System.out.println (this + " empile "
                + this.value);
            getPile().push(this.value++);
        }
    }
}
```

### Exemple (début)

Les consommateurs :




```
public class Consommateur extends Fonctionneur
{
    public Consommateur()
    { super(); }

    public Consommateur(long sleepTime)
    { super(sleepTime); }
```

**Exemple (fin)**


Les consommateurs :



```
public void run()
{
    while (true)
    {
        try
        {
            Thread.sleep ((long)Math.random()
                *this.sleepTime);
        }
        catch (InterruptedException e)
        { e.printStackTrace(); }

        System.out.println (this + " depile "
            + getPile().pop());
    }
}
```

Maintenant il ne reste plus qu'à coder une classe de démarrage afin d'initier le démarrage des *threads* à synchroniser sur la pile.

**Exemple**

```
public class Start
{
    public static void main (String args[])
    {
        // Démarrage de deux producteurs.
        new Producteur(1000);
        new Producteur(1000);

        // Démarrage de deux consommateurs.
        new Consommateur(1000);
        new Consommateur(1000);

        while (true)
        {
            // Mise en veille du thread principal.
            try
            { Thread.sleep (10000L); }
            catch (Exception e)
            { e.printStackTrace(); }
        }
    }
}
```

## 5 Les modèles d'architecture

Un **pattern** (modèle) décrit une solution connue, testée et éprouvée à un problème récurrent (ici de conception). L'objectif est de capturer la connaissance liée à ces solutions afin de pouvoir la réutiliser facilement pendant la conception de nouvelles applications.

Les **design patterns** sont des organisations de classes. La difficulté réside dans le fait de savoir identifier les moments où il faut les utiliser et les moments où cela n'est pas utile. Lorsqu'on utilise les design patterns, il faut donc pouvoir dire :

- Quand utiliser un design pattern ?
- Quel design pattern utiliser ?
- Comment le mettre en œuvre ?

Les design patterns se présentent sous la forme de diagrammes UML habituellement largement documentés. Il existe beaucoup de design patterns. Les principaux (et les plus connus et les plus usités) sont au nombre de 23 et ont été dégagés par E. Gamma, R. Helm, R. Johnson et J. Vlissides, plus connus sous le nom de « GOF » (Gang Of Four).

Ces 23 patterns sont classés en 3 catégories :

- **Les patterns de création (creational patterns)** : Abstract Factory, Builder, Factory Method, Prototype et Singleton.
- **Les patterns de structure (structural patterns)** : Adapter, Bridge, Composite, Decorator, Facade, Flyweight et Proxy.
- **Les patterns de comportement (behavioral patterns)** : Chain of Responsibilities, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method et Visitor.

Ces 23 patterns sont considérés comme étant la base de tous les autres.

### 5.1 Exemples de 2 patrons de création

#### 5.1.1 Abstract Factory

**Objectif** : fournir une interface pour la création de familles d'objets apparentés ou interdépendants sans qu'il soit nécessaire de spécifier leurs classes concrètes.

Ce pattern est en fait très utilisé lorsqu'on désire utiliser une implémentation parmi d'autres d'une famille de classes abstraites et/ou d'interfaces sans avoir à maintenir l'application si l'on change d'implémentation. C'est très souvent le cas lorsqu'on considère que les widgets graphiques peuvent être implémentés de différentes manières (Motif, Presentation Manager, ...)

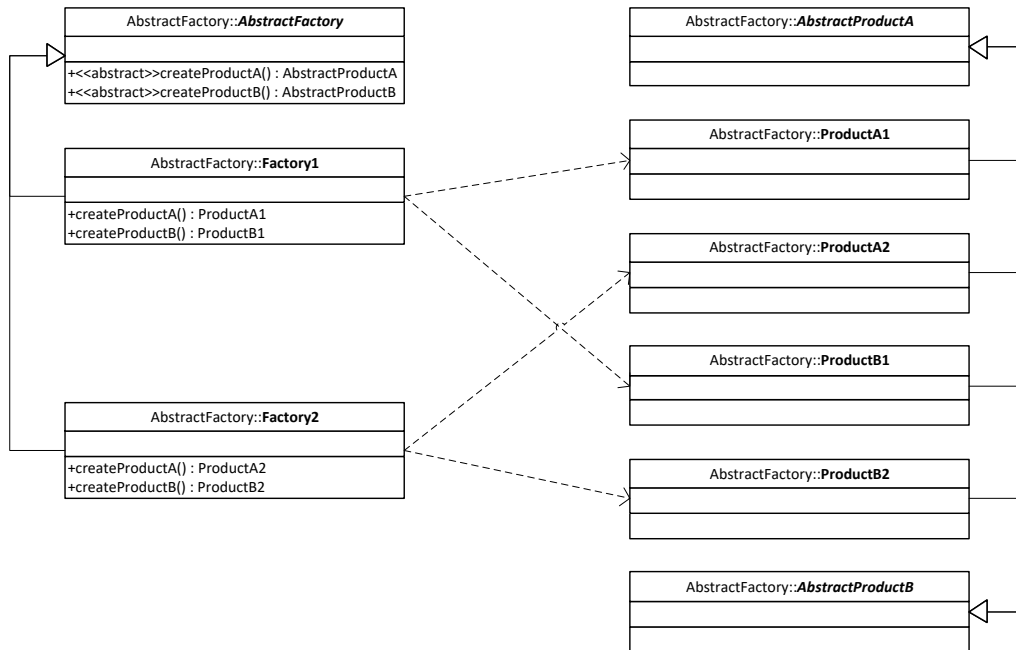


Figure 10 : diagramme de classes du pattern Abstract Factory

### Exemple (début)

Classes de gestion des fenêtres :

```

package abstractFactory;

public abstract class Fenetre
{ public abstract void afficher(); }
    
```

```

package abstractFactory;

public class FenetreMotif extends Fenetre
{
    public void afficher()
    { System.out.println("Affichage via Motif."); }
}
    
```

```

package abstractFactory;

public class FenetrePM extends Fenetre
{
    public void afficher()
    { System.out.println("Affichage via PM."); }
}
    
```



### Exemple (suite)

Classes de gestion des ascenseurs :

```
package abstractFactory;

public abstract class Ascenseur
{
    public abstract void afficher();
    public abstract void monter();
    public abstract void descendre();
}
```

```
package abstractFactory;

public class AscenseurMotif extends Ascenseur
{
    public void afficher()
    { System.out.println("Affichage via Motif."); }

    public void monter()
    { System.out.println("Montée ascenseur Motif."); }

    public void descendre()
    { System.out.println("Descente ascenseur Motif."); }
}
```

```
package abstractFactory;

public class AscenseurPM extends Ascenseur
{
    public void afficher()
    { System.out.println("Affichage via PM."); }

    public void monter()
    { System.out.println("Montée ascenseur PM."); }

    public void descendre()
    { System.out.println("Descente ascenseur PM."); }
}
```



### Exemple (suite)

Classes de gestion des fabriques :

```
package abstractFactory;

public abstract class AbstractFactory
{
    public abstract Fenetre creerFenetre();
    public abstract Ascenseur creerAscenseur();
}
```

```
package abstractFactory;

public class FabriqueMotif extends AbstractFactory
{
    public FenetreMotif creerFenetre()
    { return (new FenetreMotif()); }

    public AscenseurMotif creerAscenseur()
    { return (new AscenseurMotif()); }
}
```

```
package abstractFactory;

public class FabriquePM extends AbstractFactory
{
    public FenetrePM creerFenetre()
    { return (new FenetrePM()); }

    public AscenseurPM creerAscenseur()
    { return (new AscenseurPM()); }
}
```



### Exemple (fin)

Classe cliente :



```
package abstractFactory;

public class Client
{
    public static void UtiliserWidgets
        (AbstractFactory factory)
    {
        System.out.println("Création d'une fenêtre.");
        Fenetre fenetre = factory.creerFenetre();
        System.out.println("Affichage de la fenêtre.");
        fenetre.afficher();
        System.out.println("Création d'un ascenseur.");
        Ascenseur ascenseur = factory.creerAscenseur();
        System.out.println("Affiche de l'ascenseur.");
        ascenseur.afficher();
        System.out.println("Montée de l'ascenseur.");
        ascenseur.monter();
        System.out.println("Descente de l'ascenseur.");
        ascenseur.descendre();
    }

    public static void main(String[] args)
    {
        System.out.println("On utilise Motif.");
        AbstractFactory factory = new FabriqueMotif();
        Client.UtiliserWidgets(factory);
        System.out.println("-----");
        System.out.println("On utilise ensuite PM.");
        factory = new FabriquePM();
        Client.UtiliserWidgets(factory);
    }
}
```

## 5.1.2 Singleton

**Objectif :** on veut s'assurer qu'une seule et unique instance d'une classe est créée pendant toute la durée de l'application. Ce pattern sera par exemple utilisé dans le cas d'un objet de connexion à une base de données.

Le design pattern **Singleton** est très facile à utiliser mais beaucoup de personnes l'utilisent mal.

Pour s'assurer de l'unicité de l'instance du singleton, il faut d'abord penser à limiter les accès aux constructeurs : le plus souvent **protected** ou **private** (attention, dans le premier cas on ne pourra pas vraiment assurer l'unicité de l'instance si certaines classes héritent de la classe **Singleton**). Puisqu'il n'y a alors aucun constructeur public, il faut donc un autre moyen pour retourner une instance de la classe **Singleton** : implémenter une méthode **public** et **static** permet cela. Par convention elle s'appellera **getInstance()**.

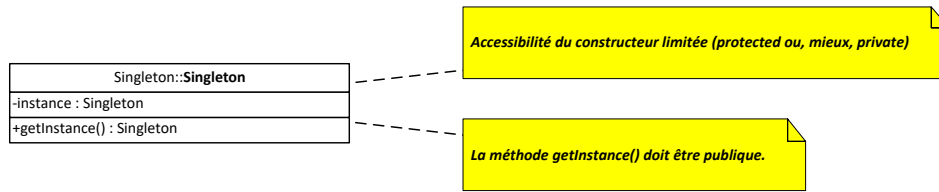


Figure 11 : diagramme de classes du pattern Singleton

### Exemple

Classe du Singleton :

```

package singleton;

public class Singleton
{
    private static Singleton instance = null;

    public static Singleton getInstance()
    {
        if (instance == null)
            instance = new Singleton();
        return (instance);
    }

    private Singleton()
    {}
}
  
```

Classe cliente :

```

package singleton;

public class Client
{
    public static void main(String[] args)
    {
        Singleton s1 = Singleton.getInstance();
        Singleton s2 = Singleton.getInstance();

        if (s1 == s2)
            System.out.println("Les 2 variables désignent  
bien la même instance.");
        else
            System.out.println("Les 2 variables ne désignent  
pas la même instance.");
    }
}
  
```



## 5.2 Exemples de 2 patrons de structure

### 5.2.1 Adapter

**Objectif** : pouvoir convertir l'interface d'une classe en une autre conforme à l'attente du client.

Le design pattern **Adapter** permet ainsi à des classes de collaborer alors qu'elles n'auraient naturellement pas pu le faire du fait d'interfaces incompatibles. L'idée est donc globalement d'adapter une implémentation à une autre afin de généraliser les API.

La structure de la solution offerte par ce pattern est assez simple : elle redéfinit les méthodes d'accès aux API. L'interface **Adapter** permet ainsi de définir l'ensemble des méthodes qui seront communes à toutes les classes à adapter. Les implémentations de l'interface **Adapter** font le lien entre les API et la normalisation que l'on souhaite mettre en place.

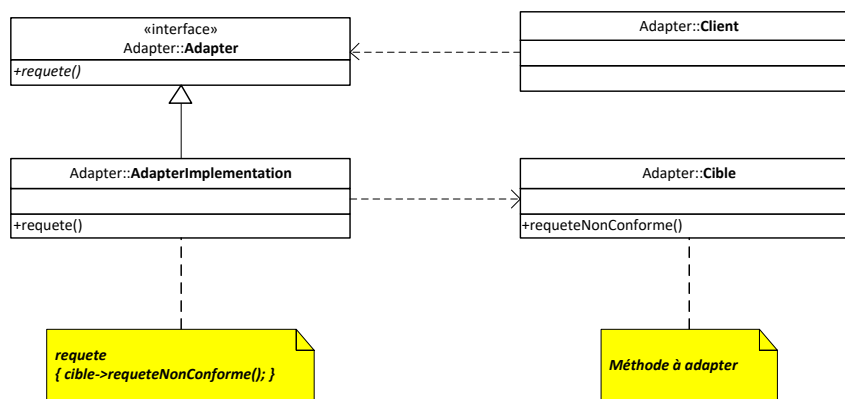


Figure 12 : diagramme de classes du pattern Adapter

**Illustration** : on a à disposition des API d'envoi de méls (**APIMail1** et **APIMail2**). Un **Adapter** permettra d'utiliser l'API que l'on veut très facilement. Par exemple, ici, on veut que la méthode standard d'envoi de méls soit **envoyerMel()**. On a donc besoin d'un **Adapter**.

#### Exemple (début)

Classes définissant les cibles à adapter (pour unifier les méthodes d'envoi de méls) :

```

package adapter.APIMail1;

public class APIMailBase
{
    public APIMailBase()
    {}

    public static boolean envoiMel()
    {
        System.out.println("Envoi de mél avec APIMail1.");
        return (true);
    }
}

```



### Exemple (suite)

```
package adapter.APIMail2;

public class Base
{
    public Base()
    {}

    public static boolean sendMail()
    {
        System.out.println("Envoi de mél avec APIMail2.");
        return (true);
    }
}
```

Classes définissant l'adapter et ses 2 implémentations :

```
package adapter;

public interface APIMailAdapter
{ public boolean envoyerMel(); }
```

```
package adapter;

import adapter.APIMail1.APIMailBase;

public class APIMail1AdapterImplementation
    implements APIMailAdapter
{
    public boolean envoyerMel()
    { return (APIMailBase.envoiMel()); }
}
```

```
package adapter;

import adapter.APIMail2.Base;

public class APIMail2AdapterImplementation
    implements APIMailAdapter
{
    public boolean envoyerMel()
    { return (Base.sendMail()); }
}
```



**Exemple (fin)**

Classe cliente :



```
package adapter;

public class Client
{
    protected APIMailAdapter api;

    public Client(APIMailAdapter api)
    { this.api = api; }

    public APIMailAdapter getAPI()
    { return (this.api); }

    public static void main(String[] args)
    {
        Client clientAvecAPIMail1 =
            new Client(new APIMail1AdapterImplementation());
        Client clientAvecAPIMail2 =
            new Client(new APIMail2AdapterImplementation());

        clientAvecAPIMail1.getAPI().envoyerMel();
        clientAvecAPIMail2.getAPI().envoyerMel();
    }
}
```

**5.2.2 Composite**

**Objectif :** on veut pouvoir représenter des hiérarchies complexes (*i.e.* récursives) de façon à n'avoir à distinguer que le moins possibles les nœuds et les feuilles (manipulation la plus uniforme possibles des composants de la hiérarchie).

L'exemple usuel de l'utilisation de ce pattern est la représentation d'une arborescence de fichiers. Un cas réel fréquent d'utilisation de ce pattern est celui du composant **JPanel** de Swing : celui-ci peut contenir d'autres composants ( **JButton** ,  **JLabel** , ...) mais également d'autres  **JPanel** .

La structure est la même que celle d'un arbre, elle se compose :

- D'une classe abstraite **Component** qui définit les méthodes communes aux nœuds et aux feuilles de la hiérarchie,
- D'une classe d'implémentation **Leaf** qui représente les feuilles,
- D'une classe d'implémentation **Composite** qui représente les nœuds de l'arbre.

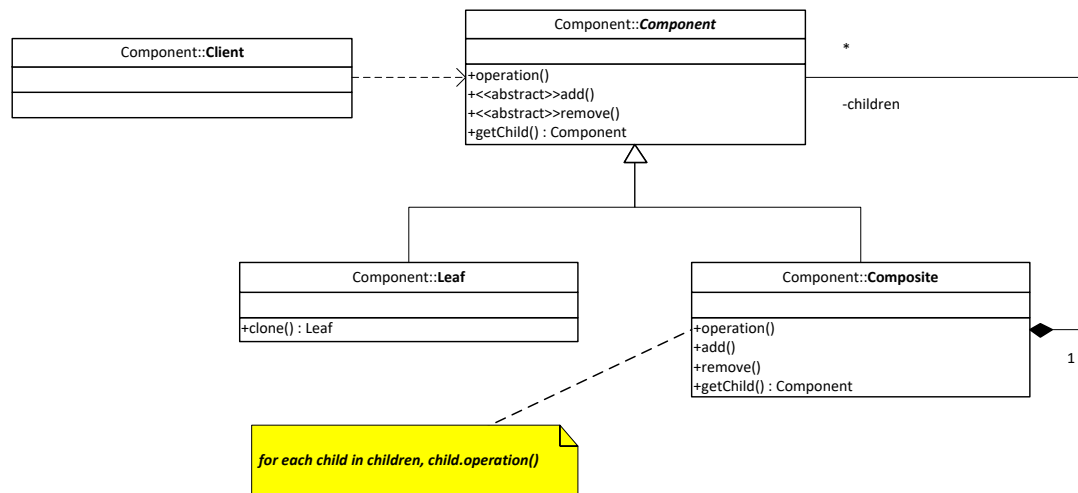


Figure 13 : diagramme de classes du pattern Composite

### Exemple (début)

Classe définissant un composant (notion abstraite) :

```

package composite;

public abstract class Component
{
    protected String name;

    public Component(String name)
    { this.name = name; }

    public abstract boolean add(Component c);

    public abstract boolean remove(Component c);

    public boolean display(int indentation)
    {
        StringBuffer strBuf = new StringBuffer();

        for (int i = 0; i < indentation; i++)
            strBuf.append("-");

        System.out.println(strBuf.toString()
            + "+" + this.name);

        return (true);
    }
}

```



### Exemple (suite)

Classes définissant les feuilles et les composites :

```
package composite;

public class Leaf extends Component
{
    public Leaf(String name)
    { super(name); }

    public boolean add(Component c)
    {
        System.err.println("Impossible de rajouter des
            éléments dans une feuille !");
        return (false);
    }

    public boolean remove(Component c)
    {
        System.err.println("Impossible de supprimer des
            éléments dans une feuille : elle n'en contient
            pas !");
        return (false);
    }
}
```

```
package composite;

import java.util.ArrayList;

public class Composite extends Component
{
    private ArrayList<Component> elements;

    public Composite(String name)
    {
        super(name);
        elements = new ArrayList<Component>();
    }

    public boolean add(Component c)
    {
        elements.add(c);
        return (true);
    }

    public boolean remove(Component c)
    {
        elements.remove(c);
        return (true);
    }
}
```



### Exemple (fin)

```
public boolean display(int indentation)
{
    super.display(indentation);
    for (int i = 0; i < elements.size(); i++)
        ((Component)elements.get(i))
            .display(indentation + 1);

    return (true);
}
```

Classe cliente :

```
package composite;

public class Client
{
    public static void main(String[] args)
    {
        Composite root = new Composite("Racine");
        Composite sousRoot1 = new Composite("SousRoot1");
        Composite sousSousRoot =
            new Composite("SousSousRoot");
        Composite sousRoot2 = new Composite("SousRoot2");

        sousSousRoot.add(new Leaf("SSElement1"));
        sousSousRoot.add(new Leaf("SSElement2"));
        sousSousRoot.add(new Leaf("SSElement3"));

        sousRoot1.add(sousSousRoot);

        sousRoot2.add(new Leaf("SElement1"));
        sousRoot2.add(new Leaf("SElement2"));
        sousRoot2.add(new Leaf("SElement3"));

        root.add(new Leaf("RacineElement1"));
        root.add(sousRoot1);
        root.add(new Leaf("RacineElement2"));
        root.add(sousRoot2);

        root.display(0);
    }
}
```



## 5.3 Exemples de 2 patrons de comportement

### 5.3.1 Iterator

**Objectif :** permettre d'accéder séquentiellement aux éléments d'un agrégat.

L'idée est de pouvoir parcourir et accéder aux éléments sans pour autant exposer la structure interne de l'agrégat. La possibilité de parcourir le contenu de différentes façons (ordre croissant, ordre décroissant, ...) est également offerte.

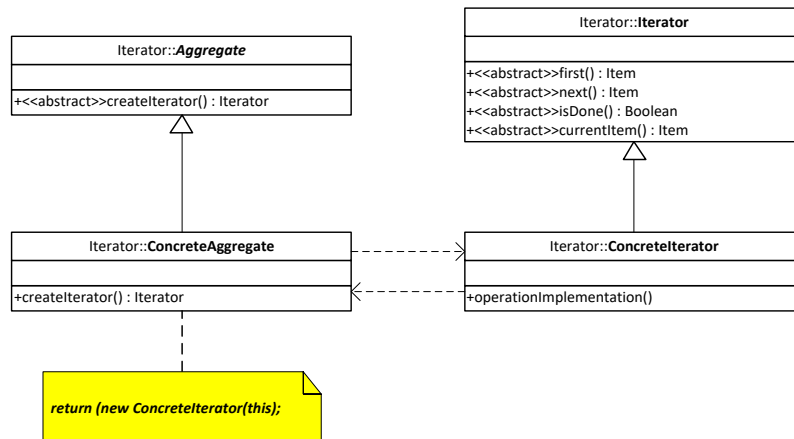


Figure 14 : diagramme de classes du pattern Iterator

#### Exemple (début)

Interfaces de collections (agrégats) et d'itérateurs :

```

package iterator.abstracts;

public interface ICollection
{
    public IIterator createIterator();
    public Object getElement(int index);
    public int getIndexMax();
    public void add(Object item);
    public void remove(Object item);
}

```

```

package iterator.abstracts;

public interface IIterator
{
    public Object first();
    public Object next();
    public boolean isDone();
    public Object current();
}

```



### Exemple (suite)

Classe définissant une collection :

```
package iterator;

import iterator.abstracts ICollection;
import iterator.abstracts IIterator;
import java.util.ArrayList;

public class CollectionSimple implements ICollection
{
    private ArrayList<Object> array;

    public CollectionSimple()
    { this.array = new ArrayList<Object>(); }

    public IIterator createIterator()
    { return (new IteratorSimple(this)); }

    public IteratorWithStep createIterator(int step)
    { return (new IteratorWithStep(this, step)); }

    public IteratorWithCondition createIterator
        (String conditionStartWith)
    {
        return (new IteratorWithCondition
            (this, conditionStartWith));
    }

    public Object getElement(int index)
    {
        if (this.array.size() > index)
            return (this.array.get(index));
        else
            return (null);
    }

    public int getIndexMax()
    { return (this.array.size()); }

    public void add(Object item)
    { this.array.add(item); }

    public void remove(Object item)
    { this.array.remove(item); }
}
```





### Exemple (suite)

Classe définissant un itérateur :

```
package iterator;

import iterator.abstracts ICollection;
import iterator.abstracts IIterator;

public class IteratorSimple implements IIterator
{
    protected ICollection collection;
    private int current;

    public IteratorSimple(ICollection collection)
    {
        this.collection = collection;
        this.current = -1;
    }

    public Object first()
    {
        this.current = 0;
        return (this.collection.getElement(this.current));
    }

    public Object next()
    {
        this.current++;
        if (!this.isDone())
            return (this.collection.getElement(this.current));
        else
            return (null);
    }

    public boolean isDone()
    {
        return (this.current >=
            this.collection.getIndexMax());
    }

    public Object current()
    { return (this.collection.getElement(this.current)); }
}
```



### Exemple (suite)

Classe définissant un deuxième itérateur :

```
package iterator;

import iterator.abstracts ICollection;
import iterator.abstracts IIterator;

public class IteratorWithStep implements IIterator
{
    protected ICollection collection;
    private int current;
    private int step;

    public IteratorWithStep(ICollection collection)
    {
        this.collection = collection;
        this.current = -1;
        this.step = 1;
    }

    public IteratorWithStep(ICollection collection,
        int step)
    {
        this.collection = collection;
        this.current = -step;
        this.step = step;
    }

    public Object first()
    {
        this.current = 0;
        return (this.collection.getElement(this.current));
    }

    public Object next()
    {
        this.current += step;
        if (!this.isDone()) return
            (this.collection.getElement(this.current));
        else return (null);
    }

    public boolean isDone()
    {
        return (this.current >=
            this.collection.getIndexMax());
    }

    public Object current()
    {
        return (this.collection.getElement(this.current));
    }
}
```



### Exemple (suite)

Classe définissant un troisième itérateur :

```
package iterator;

import iterator.abstracts ICollection;
import iterator.Abstracts.IIterator;

public class IteratorWithCondition implements IIterator
{
    protected ICollection collection;
    private int current;
    protected String condition;

    public IteratorWithCondition(ICollection collection)
    {
        this.collection = collection;
        this.current = -1;
        this.condition = "";
    }

    public IteratorWithCondition
        (ICollection collection, String condition)
    {
        this.collection = collection;
        this.current = -1;
        this.condition = condition;
    }

    public Object first()
    {
        this.current = 0;
        return (this.collection.getElement(this.current));
    }

    public Object next()
    {
        this.current++;
        if (!this.isDone())
            if (this.collection.getElement(this.current)
                .toString().startsWith(this.condition))
                return (this.collection.getElement
                    (this.current));
            else
                return (this.next());
        else
            return (null);
    }
}
```



### Exemple (suite)

```
public boolean isDone()
{
    return (this.current >=
            this.collection.getIndexMax());
}

public Object current()
{
    return (this.collection.getElement(this.current));
}
}
```

Classe définissant un item de la collection :



```
package iterator;

public class Item
{
    protected String texte;

    public Item(String texte)
    { this.texte = texte; }

    public void setTexte(String texte)
    { this.texte = texte; }

    public String getTexte()
    { return (this.texte); }

    public String toString()
    { return (this.texte); }
}
```

### Exemple (fin)

Classe cliente :

```
package iterator;

public class ClientIterator
{
    public static void main(String[] args)
    {
        CollectionSimple listItems = new CollectionSimple();
        listItems.add(new Item("Item1"));
        listItems.add(new Item("AItem2"));
        listItems.add(new Item("Item3"));
        listItems.add(new Item("Item4"));
        listItems.add(new Item("AItem5"));
        listItems.add(new Item("Item6"));
        listItems.add(new Item("Item7"));
        listItems.add(new Item("Item8"));

        IteratorSimple iterator =
            (IteratorSimple)listItems.createIterator();
        IteratorWithStep iteratorStep =
            (IteratorWithStep)listItems.createIterator(2);
        IteratorWithCondition iteratorCondition =
            (IteratorWithCondition)listItems.createIterator
                ("AItem");

        Item itemTmp;

        System.out.println("-- ITERATOR SIMPLE --");
        while ((itemTmp = (Item)iterator.next()) != null)
            System.out.println(itemTmp);

        System.out.println("-- ITERATOR WITH STEP 2 --");
        while ((itemTmp = (Item)iteratorStep.next())
            != null)
            System.out.println(itemTmp);

        System.out.println("-- ITERATOR WITH START
            CONDITION 'AItem' --");
        while ((itemTmp = (Item)iteratorCondition.next())
            != null)
            System.out.println(itemTmp);
    }
}
```



### 5.3.2 State

**Objectif :** ce pattern permet de faire varier le comportement d'un objet en fonction de son état. En effet, la gestion du changement d'état d'un objet peut parfois poser des problèmes et ce pattern permet de les pallier de façon simple et rapide. Cependant, il n'est pas obligatoire de l'utiliser dans toutes les situations : il faudra identifier s'il s'agit réellement de changements d'états à gérer ou non. Une façon de s'en apercevoir est d'avoir des méthodes constituées de parties conditionnelles de grandes tailles (**if** ou **switch**).

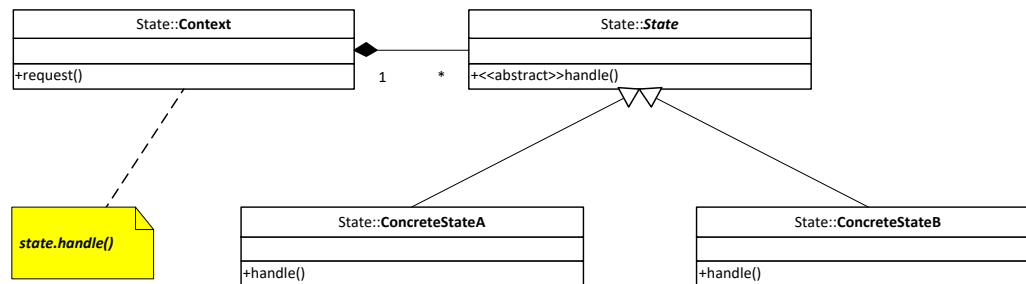


Figure 15 : diagramme de classes du pattern State

La classe **Context** comporte un attribut de type **State** et une méthode **request()** qui change cet état. La classe **State** est abstraite afin de forcer les classes qui la dérivent à redéfinir la méthode abstraite **handle()**. La classe **Client** permet d'instancier un objet de type **Context** et d'en afficher l'état suite à des changements liés à l'appel à la méthode **request()**.

#### Exemple (début)

Classe gérant le contexte :



```

package state;

public class Context
{
    private State state;

    public Context(State state)
    { this.state = state; }

    public void request()
    { state.handle(this); }

    public void show()
    { System.out.println("State : " + state); }

    public void setState(State state)
    { this.state = state; }
}
  
```

### Exemple (fin)

Classes gérant les états :

```
package state;

abstract class State
{ abstract public void handle(Context context); }
```

```
package state;

public class ConcreteStateA extends State
{
    public void handle(Context context)
    { context.setState(new ConcreteStateB()); }
}
```

```
package state;

public class ConcreteStateB extends State
{
    public void handle(Context context)
    { context.setState(new ConcreteStateA()); }
}
```

Classe cliente :

```
package state;

public class Client
{
    public static void main(String[] args)
    {
        Context c = new Context(new ConcreteStateA());
        c.show();

        c.request();
        c.show();

        c.request();
        c.show();
    }
}
```







## Table des illustrations

Figure 1 : classes prenant en charge les manipulations de flux binaires .....	11
Figure 2 : classes prenant en charge les manipulations de flux de caractères .....	16
Figure 3 : mécanismes de sérialisation dans un flux .....	25
Figure 4 : mécanisme de sérialisation dans une BD relationnelle.....	25
Figure 5 : demande de connexion en mode TCP d'un client à un serveur .....	33
Figure 6 : suite de la communication en mode TCP entre le client et le serveur .....	33
Figure 7 : principe de la réification.....	45
Figure 8 : threads d'un processus Java faisant une boucle infinie ( <code>while (true)</code> ) .....	52
Figure 9 : cycle de vie d'un <i>thread</i> .....	62
Figure 10 : diagramme de classes du pattern Abstract Factory .....	84
Figure 11 : diagramme de classes du pattern Singleton .....	88
Figure 12 : diagramme de classes du pattern Adapter .....	89
Figure 13 : diagramme de classes du pattern Composite .....	92
Figure 14 : diagramme de classes du pattern Iterator .....	95
Figure 15 : diagramme de classes du pattern State .....	102